# A Byte of Python

**Swaroop C H**

# A Byte of Python

Swaroop C H

## Abstract

This book will help you to learn the Python programming language, whether you are new to computers or are an experienced programmer.

# Table of Contents

# List of Tables

# List of Examples

# Preface

Python is probably one of the few programming languages which is both simple and powerful. This is good for both and beginners as well as experts, and more importantly, is fun to program with. This book aims to help you learn this wonderful language and show how to get things done quickly and painlessly - in effect 'The Perfect Anti-venom to your programming problems'.

## Who This Book Is For

This book serves as a guide or tutorial to the Python programming language. It is mainly targeted at newbies. It is useful for experienced programmers as well.

The aim is that if all you know about computers is how to save text files, then you can learn Python from this book. If you have previous programming experience, then you can also learn Python from this book.

If you do have previous programming experience, you will be interested in the differences between Python and your favorite programming language - I have highlighted many such differences. A little warning though, Python is soon going to become your favorite programming language!

## History Lesson

I first started with Python when I needed to write an installer for my software Diamond [http://www.g2swaroop.net/software/] so that I could make the installation easy. I had to choose between Python and Perl bindings for the Qt library. I did some research on the web and I came across an article where Eric S. Raymond, the famous and respected hacker, talked about how Python has become his favorite programming language. I also found out that the PyQt bindings were very good compared to Perl-Qt. So, I decided that Python was the language for me.

Then, I started searching for a good book on Python. I couldn't find any! I did find some O'Reilly books but they were either too expensive or were more like a reference manual than a guide. So, I settled for the documentation that came with Python. However, it was too brief and small. It did give a good idea about Python but was not complete. I managed with it since I had previous programming experience, but it was unsuitable for newbies.

About six months after my first brush with Python, I installed the (then) latest Red Hat 9.0 Linux and I was playing around with KWord. I got excited about it and suddenly got the idea of writing some stuff on Python. I started writing a few pages but it quickly became 30 pages long. Then, I became serious about making it more useful in a book form. After a *lot* of rewrites, it has reached a stage where it has become a useful guide to learning the Python language. I consider this book to be my contribution and tribute to the open source community.

This book started out as my personal notes on Python and I still consider it in the same way, although I've taken a lot of effort to make it more palatable to others :)

In the true spirit of open source, I have received lots of constructive suggestions, criticisms and feedback from enthusiastic readers which has helped me improve this book a lot.

## Status of the book

This book is a **work-in-progress**. Many chapters are constantly being changed and improved. However, the book has matured a lot. You should be able to learn Python easily from this book. Please do tell me if you find any part of the book to be incorrect or incomprehensible.

More chapters are planned for the future, such as on wxPython, Twisted and maybe even Boa Constructor.

## Official Website

The official website of the book is www.byteofpython.info [http://www.byteofpython.info] . From the website, you can read the whole book online or you can download the latest versions of the book, and also send me feedback.

## License Terms

This book is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License 2.0 [http://creativecommons.org/licenses/by-nc-sa/2.0/] .

Basically, you are free to copy, distribute, and display the book, as long as you give credit to me. The restrictions are that you cannot use the book for commercial purposes without my permission. You are free to modify and build upon this work, provided that you clearly mark all changes and release the modified work under the same license as this book.

Please visit the Creative Commons website [http://creativecommons.org/licenses/by-nc-sa/2.0/] for the full and exact text of the license, or for an easy-to-understand version. There is even a comic strip explaining the terms of the license.

## Feedback

I have put in a lot of effort to make this book as interesting and as accurate as possible. However, if you find some material to be inconsistent or incorrect, or simply needs improvement, then please do inform me, so that I can make suitable improvements. You can reach me at `<swaroop@byteofpython.info>` .

## Something To Think About

> There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies.
>
> —C. A. R. Hoare

> Success in life is a matter not so much of talent and opportunity as of concentration and perseverance.
>
> —C. W. Wendte

# Chapter 1. Introduction

## Introduction

Python is one of those rare languages which can claim to be both **simple** and **powerful**. You will find that you will be pleasantly surprised on how easy it is to concentrate on the solution to the problem rather than the syntax and structure of the language you are programming in.

The official introduction to Python is

> Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

I will discuss most of these features in more detail in the next section.

### Note

Guido van Rossum, the creator of the Python language, named the language after the BBC show "Monty Python's Flying Circus ". He doesn't particularly like snakes that kill animals for food by winding their long bodies around them and crushing them.

## Features of Python

| | |
|---|---|
| Simple | Python is a simple and minimalistic language. Reading a good Python program feels almost like reading English, although very strict English! This pseudo-code nature of Python is one of its greatest strengths. It allows you to concentrate on the solution to the problem rather than the language itself. |
| Easy to Learn | As you will see, Python is extremely easy to get started with. Python has an extraordinarily simple syntax, as already mentioned. |
| Free and Open Source | Python is an example of a FLOSS (Free/Libré and Open Source Software). In simple terms, you can freely distribute copies of this software, read it's source code, make changes to it, use pieces of it in new free programs, and that you know you can do these things. FLOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and is constantly improved by a community who just want to see a better Python. |
| High-level Language | When you write programs in Python, you never need to bother about the low-level details such as managing the memory used by your program, etc. |
| Portable | Due to its open-source nature, Python has been ported (i.e. changed to make it work on) to many platforms. All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features. <br><br> You can use Python on Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, |

Windows CE and even PocketPC !

Interpreted

This requires a bit of explanation.

A program written in a compiled language like C or C++ is converted from the source language i.e. C or C++ into a language that is spoken by your computer (binary code i.e. 0s and 1s) using a compiler with various flags and options. When you run the program, the linker/loader software copies the program from hard disk to memory and starts running it.

Python, on the other hand, does not need compilation to binary. You just *run* the program directly from the source code. Internally, Python converts the source code into an intermediate form called bytecodes and then translates this into the native language of your computer and then runs it. All this, actually, makes using Python much easier since you don't have to worry about compiling the program, making sure that the proper libraries are linked and loaded, etc, etc. This also makes your Python programs much more portable, since you can just copy your Python program onto another computer and it just works!

Object Oriented

Python supports procedure-oriented programming as well as object-oriented programming. In *procedure-oriented* languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In *object-oriented* languages, the program is built around objects which combine data and functionality. Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.

Extensible

If you need a critical piece of code to run very fast or want to have some piece of algorithm not to be open, you can code that part of your program in C or C++ and then use them from your Python program.

Embeddable

You can embed Python within your C/C++ programs to give 'scripting' capabilities for your program's users.

Extensive Libraries

The Python Standard Library is huge indeed. It can help you do various things involving regular expressions, documentation generation, unit testing, threading, databases, web browsers, CGI, ftp, email, XML, XML-RPC, HTML, WAV files, cryptography, GUI (graphical user interfaces), Tk, and other system-dependent stuff. Remember, all this is always available wherever Python is installed. This is called the 'Batteries Included' philosophy of Python.

Besides, the standard library, there are various other high-quality libraries such as wxPython [http://www.wxpython.org], Twisted [http://www.twistedmatrix.com/products/twisted], Python Imaging Library [http://www.pythonware.com/products/pil/index.htm] and many more.

## Summary

Python is indeed an exciting and powerful language. It has the right combination of performance and features that make writing programs in Python both fun and easy.

## Why not Perl?

If you didn't know already, Perl is another extremely popular open source interpreted programming language.

If you have ever tried writing a large program in Perl, you would have answered this question yourself! In other words, Perl programs are easy when they are small and it excels at small hacks and scripts to 'get work done'. However, they quickly become unwieldy once you start writing bigger programs and I am speaking this out of experience of writing large Perl programs at Yahoo!

When compared to Perl, Python programs are definitely simpler, clearer, easier to write and hence more understandable and maintainable. I do admire Perl and I do use it on a daily basis for various things but whenever I write a program, I always start thinking in terms of Python because it has become so natural for me. Perl has undergone so many hacks and changes, that it feels like it is one big (but one hell of a) hack. Sadly, the upcoming Perl 6 does not seem to be making any improvements regarding this.

The only and very significant advantage that I feel Perl has, is its huge CPAN [http://cpan.perl.org] library - the Comprehensive Perl Archive Network. As the name suggests, this is a humongous collection of Perl modules and it is simply mind-boggling because of its sheer size and depth - you can do virtually anything you can do with a computer using these modules. One of the reasons that Perl has more libraries than Python is that it has been around for a much longer time than Python. Maybe I should suggest a port-Perl-modules-to-Python hackathon on comp.lang.python [http://groups.google.com/groups?q=comp.lang.python] :)

Also, the new Parrot virtual machine [http://www.parrotcode.org] is designed to run both the completely redesigned Perl 6 as well as Python and other interpreted languages like Ruby, PHP and Tcl. What this means to you is that *maybe* you will be able to use all Perl modules from Python in the future, so that will give you the best of both worlds - the powerful CPAN library combined with the powerful Python language. However, we will have to just wait and see what happens.

## What Programmers Say

You may find it interesting to read what great hackers like ESR have to say about Python:

- **Eric S. Raymond** is the author of 'The Cathedral and the Bazaar' and is also the person who coined the term 'Open Source'. He says that Python has become his favorite programming language [http://www.linuxjournal.com/article.php?sid=3882]. This article was the real inspiration for my first brush with Python.

- **Bruce Eckel** is the author of the famous 'Thinking in Java' and 'Thinking in C++' books. He says that no language has made him more productive than Python. He says that Python is perhaps the only language that focuses on making things easier for the programmer. Read the complete interview [http://www.artima.com/intv/aboutme.html] for more details.

- **Peter Norvig** is a well-known Lisp author and Director of Search Quality at Google (thanks to Guido van Rossum for pointing that out). He says that Python has always been an integral part of Google. You can actually verify this statement by looking at the Google Jobs [http://www.google.com/jobs/index.html] page which lists Python knowledge as a requirement for software engineers.

- **Bruce Perens** is a co-founder of OpenSource.org and the UserLinux project. UserLinux aims to create a standardized Linux distribution supported by multiple vendors. Python has beaten contenders like Perl and Ruby to become the main programming language that will be supported by UserLinux.

# Chapter 2. Installing Python

## For Linux/BSD users

If you are using a Linux distribution such as Fedora or Mandrake or {put your choice here}, or a BSD system such as FreeBSD, then you probably already have Python installed on your system.

To test if you have Python already installed on your Linux box, open a shell program (like konsole or gnome-terminal) and enter the command **python -V** as shown below.

```
$ python -V
Python 2.3.4
```

### Note

`$` is the prompt of the shell. It will be different for you depending on the settings of your OS, hence I will indicate the prompt by just the `$` symbol.

If you see some version information like the one shown above, then you have Python installed already.

However, if you get a message like this one:

```
$ python -V
bash: python: command not found
```

then, you don't have Python installed. This is highly unlikely but possible.

In this case, you have two ways of installing Python on your system.

- Install the binary packages using the package management software that comes with your OS, such as yum in Fedora Linux, urpmi in Mandrake Linux, apt-get in Debian Linux, pkg_add in FreeBSD, etc. Note that you will need an internet connection to use this method.

  Alternatively, you can download the binaries from somewhere else and then copy to your PC and install it.

- You can compile Python from the source code [http://www.python.org/download/] and install it. The compilation instructions are provided at the website.

## For Windows Users

Visit Python.org/download [http://www.python.org/download/] and download the latest version from this website (which was 2.3.4 [http://www.python.org/ftp/python/2.3.4/Python-2.3.4.exe] as of this writing. This is just 9.4 MB which is very compact compared to most other languages. The installation is just like any other Windows-based software.

### Caution

When you are given the option of unchecking any *optional* components, don't uncheck any! Some of these components can be useful for you, especially IDLE.

An interesting fact is that about 70% of Python downloads are by Windows users. Of course, this doesn't give the complete picture since almost all Linux users will have Python installed already on their systems by default.

### Using Python in the Windows command line

If you want to be able to use Python from the Windows command line, then you need to set the PATH variable appropriately.

For Windows 2000, XP, 2003 , click on Control Panel -> System -> Advanced -> Environment Variables. Click on the variable named **PATH** in the 'System Variables' section, then select Edit and add **;C:\Python23** (without the quotes) to the end of what is already there. Of course, use the appropriate directory name.

For older versions of Windows, add the following line to the file `C:\AUTOEXEC.BAT` : '**PATH=%PATH%;C:\Python23**' (without the quotes) and restart the system. For Windows NT, use the `AUTOEXEC.NT` file.

## Summary

For a Linux system, you most probably already have Python installed on your system. Otherwise, you can install it using the package management software that comes with your distribution. For a Windows system, installing Python is as easy as downloading the installer and double-clicking on it. From now on, we will assume that you have Python installed on your system.

Next, we will write our first Python program.

# Chapter 3. First Steps

## Introduction

We will now see how to run a traditional 'Hello World' program in Python. This will teach you how to write, save and run Python programs.

There are two ways of using Python to run your program - using the interactive interpreter prompt or using a source file. We will now see how to use both the methods.

## Using the interpreter prompt

Start the intepreter on the command line by entering **python** at the shell prompt. Now enter `print 'Hello World'` followed by the **Enter** key. You should see the words `Hello World` as output.

For Windows users, you can run the interpreter in the command line if you have set the `PATH` variable appropriately. Alternatively, you can use the IDLE program. IDLE is short for Integrated DeveLopment Environment. Click on Start -> Programs -> Python 2.3 -> IDLE (Python GUI). Linux users can use IDLE too.

Note that the <<< signs are the prompt for entering Python statements.

**Example 3.1. Using the python interpreter prompt**

```
$ python
Python 2.3.4 (#1, Oct 26 2004, 16:42:40)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello world'
hello world
>>>
```

Notice that Python gives you the output of the line immediately! What you just entered is a single Python *statement*. We use `print` to (unsurprisingly) print any value that you supply to it. Here, we are supplying the text `Hello World` and this is promptly printed to the screen.

### How to quit the Python prompt

To exit the prompt, press **Ctrl-d** if you are using IDLE or are using a Linux/BSD shell. In case of the Windows command prompt, press **Ctrl-z** followed by **Enter**.

## Choosing an Editor

Before we move on to writing Python programs in source files, we need an editor to write the source files. The choice of an editor is crucial indeed. You have to choose an editor as you would choose a car you would buy. A good editor will help you write Python programs easily, making your journey more comfortable and helps you reach your destination (achieve your goal) in a much faster and safer way.

One of the very basic requirements is **syntax highlighting** where all the different parts of your Python program are colorized so that you can *see* your program and visualize its running.

If you are using Windows, then I suggest that you use IDLE. IDLE does syntax highlighting and a lot more such as allowing you to run your programs within IDLE among other things. A special note: **don't use Notepad** - it is a bad choice because it does not do syntax highlighting and also importantly it does not support indentation of the text which is very important in our case as we will see later. Good editors such as IDLE (and also VIM) will automatically help you do this.

If you are using Linux/FreeBSD, then you have a lot of choices for an editor. If you are an experienced programmer, then you must be already using VIM or Emacs. Needless to say, these are two of the most powerful editors and you will be benefitted by using them to write your Python programs. I personally use VIM for most of my programs. If you are a beginner programmer, then you can use Kate which is one of my favorites. In case you are willing to take the time to learn VIM or Emacs, then I highly recommend that you do learn to use either of them as it will be very useful for you in the long run.

If you still want to explore other choices of an editor, see the comprehensive list of Python editors [http://www.python.org/cgi-bin/moinmoin/PythonEditors] and make your choice. You can also choose an IDE (Integrated Development Environment) for Python. See the comprehensive list of IDEs that support Python [http://www.python.org/cgi-bin/moinmoin/IntegratedDevelopmentEnvironments] for more details. Once you start writing large Python programs, IDEs can be very useful indeed.

I repeat once again, please choose a proper editor - it can make writing Python programs more fun and easy.

## Using a Source File

Now let's get back to programming. There is a tradition that whenever you learn a new programming language, the first program that you write and run is the 'Hello World' program - all it does is just say 'Hello World' when you run it. As Simon Cozens [1] puts it, it is the 'traditional incantation to the programming gods to help you learn the language better' :) .

Start your choice of editor, enter the following program and save it as `helloworld.py`

### Example 3.2. Using a Source File

```
#!/usr/bin/python
# Filename : helloworld.py
print 'Hello World'
```

(Source file: code/helloworld.py)

Run this program by opening a shell (Linux terminal or DOS prompt) and entering the command **python *`helloworld.py`***. If you are using IDLE, use the menu Edit -> Run Script or the keyboard shortcut **Ctrl-F5**. The output is as shown below.

## Output

[1] one of the leading Perl6/Parrot hackers and the author of the amazing 'Beginning Perl' book

```
$ python helloworld.py
Hello World
```

If you got the output as shown above, congratulations! - you have successfully run your first Python program.

In case you got an error, please type the above program *exactly* as shown and above and run the program again. Note that Python is case-sensitive i.e. `print` is not the same as `Print` - note the lowercase `p` in the former and the uppercase `P` in the latter. Also, ensure there are no spaces or tabs before the first character in each line - we will see why this is important later.

## How It Works

Let us consider the first two lines of the program. These are called *comments* - anything to the right of the # symbol is a comment and is mainly useful as notes for the reader of the program.

Python does not use comments except for the special case of the first line here. It is called the *shebang line* - whenever the first two characters of the source file are `#!` followed by the location of a program, this tells your Linux/Unix system that this program should be run with this interpreter when you *execute* the program. This is explained in detail in the next section. Note that you can always run the program on any platform by specifying the interpreter directly on the command line such as the command **python *helloworld.py***.

### Important

Use comments sensibly in your program to explain some important details of your program - this is useful for readers of your program so that they can easily understand what the program is doing. Remember, that person can be yourself after six months!

The comments are followed by a Python *statement* - this just prints the text `'Hello World'`. The `print` is actually an operator and `'Hello World'` is referred to as a string - don't worry, we will explore these terminologies in detail later.

## Executable Python programs

This applies only to Linux/Unix users but Windows users might be curious as well about the first line of the program. First, we have to give the program executable permission using the **chmod** command then *run* the source program.

```
$ chmod a+x helloworld.py
$ ./helloworld.py
Hello World
```

The chmod command is used here to *ch*ange the *mod*e of the file by giving e*x*ecute permission to *a*ll users of the system. Then, we execute the program directly by specifying the location of the source file. We use the `./` to indicate that the program is located in the current directory.

To make things more fun, you can rename the file to just `helloworld` and run it as **./helloworld** and it will still work since the system knows that it has to run the program using the interpreter whose location is specified in the first line in the source file.

You are now able to run the program as long as you know the exact path of the program - but what if you wanted to be able to run the program from anywhere? You can do this by storing the program in one of the directories listed in the `PATH` environment variable. Whenever you run any program, the system looks for that program in each of the directories listed in the `PATH` environment variable and then runs that program. We can make this program available everywhere by simply copying this source file to one of the directories listed in `PATH`.

```
$ echo $PATH
/opt/mono/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/swaroop/bin
$ cp helloworld.py /home/swaroop/bin/helloworld
$ helloworld
Hello World
```

We can display the `PATH` variable using the **echo** command and prefixing the variable name by `$` to indicate to the shell that we need the value of this variable. We see that `/home/swaroop/bin` is one of the directories in the PATH variable where **swaroop** is the username I am using in my system. There will usually be a similar directory for your username on your system. Alternatively, you can add a directory of your choice to the `PATH` variable - this can be done by running **PATH=$PATH:/home/swaroop/mydir** where `'/home/swaroop/mydir'` is the directory I want to add to the `PATH` variable.

This method is very useful if you want to write useful scripts that you want to run the program anytime, anywhere. It is like creating your own commands just like **cd** or any other commands that you use in the Linux terminal or DOS prompt.

### Caution

W.r.t. Python, a program or a script or software all mean the same thing.

## Getting Help

If you need quick information about any function or statement in Python, then you can use the built-in `help` functionality. This is very useful especially when using the interpreter prompt. For example, run `help(str)` - this displays the help for the `str` class which is used to store all text (strings) that you use in your program. Classes will be explained in detail in the chapter on object-oriented programming.

### Note

Press **q** to exit the help.

Similarly, you can obtain information about almost anything in Python. Use `help()` to learn more about using `help` itself!

In case you need to get help for operators like `print`, then you need to set the `PYTHONDOCS` environment variable appropriately. This can be done easily on Linux/Unix using the **env** command.

```
$ env PYTHONDOCS=/usr/share/doc/python-docs-2.3.4/html/ python
Python 2.3.4 (#1, Oct 26 2004, 16:42:40)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on linux2
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>> help('print')
```

You will notice that I have used quotes to specify `'print'` so that Python can understand that I want to fetch help about 'print' and I am not asking it to print something.

Note that the location I have used is the location in Fedora Core 3 Linux - it may be different for different distributions and versions.

## Summary

You should now be able to write, save and run Python programs at ease. Now that you are a Python user, let's learn some more Python concepts.

# Chapter 4. The Basics

Just printing 'Hello World' is not enough, is it? You want to do more than that - you want to take some input, manipulate it and get something out of it. We can achieve this in Python using constants and variables.

## Literal Constants

An example of a literal constant is a number like `5`, `1.23`, `9.25e-3` or a string like `'This is a string'` or `"It's a string!"`. It is called a literal because it is *literal* - you use its value literally. The number `2` always represents itself and nothing else - it is a constant because its value cannot be changed. Hence, all these are referred to as literal constants.

## Numbers

Numbers in Python are of four types - integers, long integers, floating point and complex numbers.

- Examples of integers are `2` which are just whole numbers.
- Long integers are just bigger whole numbers.
- Examples of floating point numbers (or *floats* for short) are `3.23` and `52.3E-4`. The `E` notation indicates powers of 10. In this case, `52.3E-4` means `52.3 * 10`$^{-4}$.
- Examples of complex numbers are `(-5+4j)` and `(2.3 - 4.6j)`

## Strings

A string is a *sequence* of *characters*. Strings are basically just a bunch of words.

I can almost guarantee that you will be using strings in almost every Python program that you write, so pay attention to the following part. Here's how you use strings in Python:

- ### Using Single Quotes (')

  You can specify strings using single quotes such as `'Quote me on this'` . All white space i.e. spaces and tabs are preserved as-is.

- ### Using Double Quotes (")

  Strings in double quotes work exactly the same way as strings in single quotes. An example is `"What's your name?"`

- ### Using Triple Quotes (''' or """)

  You can specify multi-line strings using triple quotes. You can use single quotes and double quotes freely within the triple quotes. An example is

```
'''This is a multi-line string. This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
'''
```

- **Escape Sequences**

  Suppose, you want to have a string which contains a single quote (`'`), how will you specify this string? For example, the string is `What's your name?`. You cannot specify `'What's your name?'` because Python will be confused as to where the string starts and ends. So, you will have to specify that this single quote does not indicate the end of the string. This can be done with the help of what is called an *escape sequence*. You specify the single quote as `\'` - notice the backslash. Now, you can specify the string as `'What\'s your name?'`.

  Another way of specifying this specific string would be `"What's your name?"` i.e. using double quotes. Similarly, you have to use an escape sequence forusing a double quote itself in a double quoted string. Also, you have to indicate the backslash itself using the escape sequence `\\`.

  What if you wanted to specify a two-line string? One way is to use a triple-quoted string as shown above or you can use an escape sequence for the newline character - `\n` to indicate the start of a new line. An example is `This is the first line\nThis is the second line`. Another useful escape sequence to know is the tab - `\t`. There are many more escape sequences but I have mentioned only the most useful ones here.

  One thing to note is that in a string, a single backslash at the end of the line indicates that the string is continued in the next line, but no newline is added. For example,

  ```
  "This is the first sentence.\
  This is the second sentence."
  ```

  is equivalent to `"This is the first sentence. This is the second sentence."`

- **Raw Strings**

  If you need to specify some strings where no special processing such as escape sequences are handled, then what you need is to specify a *raw* string by prefixing `r` or `R` to the string. An example is `r"Newlines are indicated by \n"`.

- **Unicode Strings**

  Unicode is a standard way of writing international text. If you want to write text in your native language such as Hindi or Arabic, then you need to have a Unicode-enabled text editor. Similarly, Python allows you to handle Unicode text - all you need to do is prefix `u` or `U`. For example, `u"This is a Unicode string."`.

Remember to use Unicode strings when you are dealing with text files, especially when you know that the file will contain text written in languages other than English.

- **Strings are immutable**

  This means that once you have created a string, you cannot change it. Although this might seem like a bad thing, it really isn't. We will see why this is not a limitation in the various programs that we see later on.

- **String literal concatenation**

  If you place two string literals side by side, they are automatically concatenated by Python. For example, `'What\'s' 'your name?'` is automatically converted in to `"What's your name?"`.

### Note for C/C++ Programmers

There is no separate `char` data type in Python. There is no real need for it and I am sure you won't miss it.

### Note for Perl/PHP Programmers

Remember that single-quoted strings and double-quoted strings are the same - they do not differ in any way.

### Note for Regular Expression Users

Always use raw strings when dealing with regular expressions. Otherwise, a lot of backwhacking may be required. For example, backreferences can be referred to as `'\\1'` or `r'\1'`.

## Variables

Using just literal constants can soon become boring - we need some way of storing any information and manipulate them as well. This is where *variables* come into the picture. Variables are exactly what they mean - their value can vary i.e. you can store anything using a variable. Variables are just parts of your computer's memory where you store some information. Unlike literal constants, you need some method of accessing these variables and hence you give them names.

## Identifier Naming

Variables are examples of identifiers. *Identifiers* are names given to identify *something*. There are some rules you have to follow for naming identifiers:

- The first character of the identifier must be a letter of the alphabet (upper or lowercase) or an underscore ('_').
- The rest of the identifier name can consist of letters (upper or lowercase), underscores ('_') or digits (0-9).
- Identifier names are case-sensitive. For example, `myname` and `myName` are **not** the same. Note the lowercase `n` in the former and the uppercase `N` in te latter.

- Examples of *valid* identifier names are `i`, `__my_name`, `name_23` and `a1b2_c3`.
- Examples of *invalid* identifier names are `2things`, `this is spaced out` and `my-name`.

## Data Types

Variables can hold values of different types called **data types**. The basic types are numbers and strings, which we have already discussed. In later chapters, we will see how to create our own types using classes.

## Objects

Remember, Python refers to anything used in a program as an *object*. This is meant in the generic sense. Instead of saying 'the *something*', we say 'the *object*'.

### Note for Object Oriented Programming users

Python is strongly object-oriented in the sense that everything is an object including numbers, strings and even functions.

We will now see how to use variables along with literal constants. Save the following example and run the program.

### How to write Python programs

Henceforth, the standard procedure to save and run a Python program is as follows:

1. Open your favorite editor.
2. Enter the program code given in the example.
3. Save it as a file with the filename mentioned in the comment. I follow the convention of having all Python programs saved with the extension `.py`.
4. Run the interpreter with the command **python *`program.py`*** or use IDLE to run the programs. You can also use the executable method as explained earlier.

**Example 4.1. Using Variables and Literal constants**

```
# Filename : var.py

i = 5
print i
i = i + 1
print i

s = '''This is a multi-line string.
This is the second line.'''
print s
```

## Output

```
$ python var.py
5
6
This is a multi-line string.
This is the second line.
```

## How It Works

Here's how this program works. First, we assign the literal constant value `5` to the variable `i` using the assignment operator (`=`). This line is called a statement because it states that something should be done and in this case, we connect the variable name `i` to the value `5`. Next, we print the value of `i` using the `print` statement which, unsurprisingly, just prints the value of the variable to the screen.

The we add `1` to the value stored in `i` and store it back. We then print it and expectedly, we get the value `6`.

Similarly, we assign the literal string to the variable `s` and then print it.

### Note for C/C++ Programmers

Variables are used by just assigning them a value. No declaration or data type definition is needed/used.

## Logical and Physical Lines

A physical line is what you *see* when you write the program. A logical line is what Python *sees* as a single statement. Python implicitly assumes that each *physical line* corresponds to a *logical line*.

An example of a logical line is a statement like `print 'Hello World'` - if this was on a line by itself (as you see it in an editor), then this also corresponds to a physical line.

Implicitly, Python encourages the use of a single statement per line which makes code more readable.

If you want to specify more than one logical line on a single physical line, then you have to explicitly specify this using a semicolon (`;`) which indicates the end of a logical line/statement. For example,

```
i = 5
print i
```

is effectively same as

```
i = 5;
print i;
```

and the same can be written as

```
i = 5; print i;
```

or even

```
i = 5; print i
```

However, I **strongly recommend** that you stick to **writing a single logical line in a single physical line only**. Use more than one physical line for a single logical line only if the logical line is really long. The idea is to avoid the semicolon as far as possible since it leads to more readable code. In fact, I have *never* used or even seen a semicolon in a Python program.

An example of writing a logical line spanning many physical lines follows. This is referred to as **explicit line joining**.

```
s = 'This is a string. \
This continues the string.'
print s
```

This gives the output:

```
This is a string. This continues the string.
```

Similarly,

```
print \
i
```

is the same as

```
print i
```

Sometimes, there is an implicit assumption where you don't need to use a backslash. This is the case where the logical line uses parentheses, square brackets or curly braces. This is is called **implicit line joining**. You can see this in action when we write programs using lists in later chapters.

## Indentation

Whitespace is important in Python. Actually, **whitespace at the beginning of the line is important**. This is called **indentation**. Leading whitespace (spaces and tabs) at the beginning of the logical line is used to determine the indentation level of the logical line, which in turn is used to determine the grouping of statements.

This means that statements which go together **must** have the same indentation. Each such set of statements is called a **block**. We will see examples of how blocks are important in later chapters.

One thing you should remember is how wrong indentation can give rise to errors. For example:

```
i = 5
 print 'Value is', i # Error! Notice a single space at the start of the line
print 'I repeat, the value is', i
```

When you run this, you get the following error:

```
  File "whitespace.py", line 4
    print 'Value is', i # Error! Notice a single space at the start of the line
    ^
SyntaxError: invalid syntax
```

Notice that there is a single space at the beginning of the second line. The error indicated by Python tells us that the syntax of the program is invalid i.e. the program was not properly written. What this means to you is that *you cannot arbitrarily start new blocks of statements* (except for the main block which you have been using all along, of course). Cases where you can use new blocks will be detailed in later chapters such as the control flow chapter.

### How to indent

Do **not** use a mixture of tabs and spaces for the indentation as it does not work across different platforms properly. I *strongly recommend* that you use a *single tab* or *two or four spaces* for each indentation level.

Choose any of these three indentation styles. More importantly, choose one and use it **consistently** i.e. use that indentation style *only*.

## Summary

Now that we have gone through many nitty-gritty details, we can move on to more interesting stuff such as control flow statements. Be sure to become comfortable with what you have read in this chapter.

# Chapter 5. Operators and Expressions

## Introduction

Most statements (logical lines) that you write will contain **expressions**. A simple example of an expression is `2 + 3`. An expression can be broken down into operators and operands.

*Operators* are functionality that do something and can be represented by symbols such as `+` or by special keywords. Operators require some data to operate on and such data are called *operands*. In this case, `2` and `3` are the operands.

## Operators

We will briefly take a look at the operators and their usage:

### Tip

You can evaluate the expressions given in the examples using the interpreter interactively. For example, to test the expression `2 + 3`, use the interactive Python interpreter prompt:

```
>>> 2 + 3
5
>>> 3 * 5
15
>>>
```

**Table 5.1. Operators and their usage**

| Operator | Name | Explanation | Examples |
|---|---|---|---|
| + | Plus | Adds the two objects | `3 + 5` gives `8`. `'a' + 'b'` gives `'ab'`. |
| - | Minus | Either gives a negative number or gives the subtraction of one number from the other | `-5.2` gives a negative number. `50 - 24` gives `26`. |
| * | Multiply | Gives the multiplication of the two numbers or returns the string repeated that many times. | `2 * 3` gives `6`. `'la' * 3` gives `'lalala'`. |
| ** | Power | Returns x to the power of y | `3 ** 4` gives `81` (i.e. `3 * 3 * 3 * 3`) |
| / | Divide | Divide x by y | `4/3` gives `1` (division of integers gives an integer). `4.0/3` or `4/3.0` gives `1.3333333333333333` |

| **Operator** | **Name** | **Explanation** | **Examples** |
|---|---|---|---|
| // | Floor Division | Returns the floor of the quotient | `4 // 3.0` gives `1.0` |
| % | Modulo | Returns the remainder of the division | `8%3` gives `2`. `-25.5%2.25` gives `1.5` . |
| << | Left Shift | Shifts the bits of the number to the left by the number of bits specified. (Each number is represented in memory by bits or binary digits i.e. 0 and 1) | `2 << 2` gives `8`. - `2` is represented by `10` in bits. Left shifting by 2 bits gives `1000` which represents the decimal `8`. |
| >> | Right Shift | Shifts the bits of the number to the right by the number of bits specified. | `11 >> 1` gives `5` - `11` is represented in bits by `1011` which when right shifted by 1 bit gives `101` which is nothing but decimal `5`. |
| & | Bitwise AND | Bitwise AND of the numbers | `5 & 3` gives `1`. |
| \| | Bit-wise OR | Bitwise OR of the numbers | `5 \| 3` gives `7` |
| ^ | Bit-wise XOR | `5 ^ 3` gives `6` | |
| ~ | Bit-wise invert | The bit-wise inversion of x is -(x+1) | `~5` gives `-6`. |
| < | Less Than | Returns whether x is less than y. All comparison operators return 1 for true and 0 for false. This is equivalent to the special variables `True` and `False` respectively. Note the capitalization of these variables' names. | `5 < 3` gives `0` (i.e. `False`) and `3 < 5` gives `1` (i.e. `True`). Comparisons can be chained arbitrarily: 3 < 5 < 7 gives `True`. |
| > | Greater Than | Returns whether x is greater than y | `5 < 3` returns `True`. If both operands are numbers, they are first converted to a common type. Otherwise, it always returns `False`. |
| <= | Less Than or Equal To | Returns whether x is less than or equal to y | `x = 3; y = 6; x <= y` returns `True`. |
| >= | Greater Than or Equal To | Returns whether x is greater than or equal to y | x = 4; y = 3; x >= 3 returns `True`. |
| == | Equal To | Compares if the objects are equal | `x = 2; y = 2; x == y` returns `True`. `x = 'str'; y = 'stR'; x == y` returns `False`. `x = 'str'; y = 'str'; x == y` re- |

| Operator | Name | Explanation | Examples |
|---|---|---|---|
| | | | turns `True`. |
| != | Not Equal To | Compares if the objects are not equal | `x = 2; y = 3; x != y` returns `True`. |
| not | Boolean NOT | If x is `True`, it returns `False`. If x is `False`, it returns `True`. | `x = True; not y` returns `False`. |
| and | Boolean AND | `x and y` returns `False` if x is `False`, else it returns evaluation of y | `x = False; y = True; x and y` returns `False` since x is False. In this case, Python will not evaluate y since it knows that the value of the expression will has to be false (since x is False). This is called short-circuit evaluation. |
| or | Boolean OR | If x is `True`, it returns True, else it returns evaluation of y | `x = True; y = False; x or y` returns `True`. Short-circuit evaluation applies here as well. |

## Operator Precedence

If you had an expression such as `2 + 3 * 4`, is the addition done first or the multiplication? Our high school maths tells us that the multiplication should be done first - this means that the multiplication operator has higher precedence than the addition operator.

The following table gives the operator precedence table for Python, from the lowest precedence (least binding) to the highest precedence (most binding). This means that in a given expression, Python will first evaluate the operators lower in the table before the operators listed higher in the table.

The following table (same as the one in the Python reference manual) is provided for the sake of completeness. However, I advise you to use parentheses for grouping of operators and operands in order to explicitly specify the precedence and to make the program as readable as possible. For example, `2 + (3 * 4)` is definitely more clearer than `2 + 3 * 4`. As with everything else, the parentheses shold be used sensibly and should not be redundant (as in `2 + (3 + 4)`).

### Table 5.2. Operator Precedence

| Operator | Description |
|---|---|
| lambda | Lambda Expression |
| or | Boolean OR |
| and | Boolean AND |
| not x | Boolean NOT |
| in, not in | Membership tests |
| is, is not | Identity tests |
| <, <=, >, >=, !=, == | Comparisons |

| Operator | Description |
|---|---|
| \| | Bitwise OR |
| ^ | Bitwise XOR |
| & | Bitwise AND |
| <<, >> | Shifts |
| +, - | Addition and subtraction |
| *, /, % | Multiplication, Division and Remainder |
| +x, -x | Positive, Negative |
| ~x | Bitwise NOT |
| ** | Exponentiation |
| x.attribute | Attribute reference |
| x[index] | Subscription |
| x[index:index] | Slicing |
| f(arguments ...) | Function call |
| (expressions, ...) | Binding or tuple display |
| [expressions, ...] | List display |
| {key:datum, ...} | Dictionary display |
| \`expressions, ...\` | String conversion |

The operators which we have not already come across will be explained in later chapters.

Operators with the same *same precedence* are listed in the same row in the above table. For example, `+` and `-` have the same precedence.

## Order of Evaluation

By default, the operator precedence table decides which operators are evaluated before others. However, if you want to change the orer in which they are evaluated, you can use parentheses. For example, if you want addition to be evaluated before multiplication in an expression, then you can write something like `(2 + 3) * 4`.

## Associativity

Operators are usually associated from left to right i.e. operators with same precedence are evaluated in a left to right manner. For example, `2 + 3 + 4` is evaluated as `(2 + 3) + 4`. Some operators like assignment operators have right to left associativity i.e. `a = b = c` is treated as `a = (b = c)`.

# Expressions

## Using Expressions

**Example 5.1. Using Expressions**

```
#!/usr/bin/python
```

```
# Filename: expression.py

length = 5
breadth = 2

area = length * breadth
print 'Area is', area
print 'Perimeter is', 2 * (length + breadth)
```

## Output

```
$ python expression.py
Area is 10
Perimeter is 14
```

## How It Works

The length and breadth of the rectangle are stored in variables by the same name. We use these to calculate the area and perimieter of the rectangle with the help of expressions. We store the result of the expression `length * breadth` in the variable `area` and then print it using the `print` statement. In the second case, we directly use the value of the expression `2 * (length + breadth)` in the print statement.

Also, notice how Python 'pretty-prints' the output. Even though we have not specified a space between `'Area is'` and the variable `area`, Python puts it for us so that we get a clean nice output and the program is much more readable this way (since we don't need to worry about spacing in the output). This is an example of how Python makes life easy for the programmer.

# Summary

We have seen how to use operators, operands and expressions - these are the basic building blocks of any program. Next, we will see how to make use of these in our programs using statements.

# Chapter 6. Control Flow

## Introduction

In the programs we have seen till now, there has always been a series of statements and Python faithfully executes them in the same order. What if you wanted to change the flow of how it works? For example, you want the program to take some decisions and do different things depending on different situations such as printing 'Good Morning' or 'Good Evening' depending on the time of the day?

As you might have guessed, this is achieved using control flow statements. There are three control flow statements in Python - `if`, `for` and `while`.

## The if statement

The `if` statement is used to check a condition and *if* the condition is true, we run a block of statements (called the *if-block*), *else* we process another block of statements (called the *else-block*). The *else* clause is optional.

### Using the if statement

**Example 6.1. Using the if statement**

```
#!/usr/bin/python
# Filename: if.py

number = 23
guess = int(raw_input('Enter an integer : '))

if guess == number:
        print 'Congratulations, you guessed it.' # New block starts here
        print "(but you do not win any prizes!)" # New block ends here
elif guess < number:
        print 'No, it is a little higher than that' # Another block
        # You can do whatever you want in a block ...
else:
        print 'No, it is a little lower than that'
        # you must have guess > number to reach here

print 'Done'
# This last statement is always executed, after the if statement is executed
```

### Output

```
$ python if.py
Enter an integer : 50
No, it is a little lower than that
```

```
Done
$ python if.py
Enter an integer : 22
No, it is a little higher than that
Done
$ python if.py
Enter an integer : 23
Congratulations, you guessed it.
(but you do not win any prizes!)
Done
```

## How It Works

In this program, we take guesses from the user and check if it is the number that we have. We set the variable `number` to any integer we want, say `23`. Then, we take the user's guess using the `raw_input()` function. Functions are just reusable pieces of programs. We'll read more about them in the next chapter.

We supply a string to the built-in `raw_input` function which prints it to the screen and waits for input from the user. Once we enter something and press **enter**, the function returns the input which in the case of `raw_input` is a string. We then convert this string to an integer using `int` and then store it in the variable `guess`. Actually, the `int` is a class but all you need to know right now is that you can use it to convert a string to an integer (assuming the string contains a valid integer in the text).

Next, we compare the guess of the user with the number we have chosen. If they are equal, we print a success message. Notice that we use indentation levels to tell Python which statements belong to which block. This is why indentation is so important in Python. I hope you are sticking to 'one tab per indentation level' rule. Are you?

Notice how the `if` statement contains a colon at the end - we are indicating to Python that a block of statements follows.

Then, we check if the guess is less than the number, and if so, we inform the user to guess a little higher than that. What we have used here is the `elif` clause which actually combines two related `if else-if else` statements into one combined `if-elif-else` statement. This makes the program easier and reduces the amount of indentation required.

The `elif` and `else` statements must also have a colon at the end of the logical line followed by their corresponding block of statements (with proper indentation, of course)

You can have another `if` statement inside the if-block of an `if` statement and so on - this is called a nested `if` statement.

Remember that the `elif` and `else` parts are optional. A minival valid `if` statement is

```
if True:
        print 'Yes, it is true'
```

After Python has finished executing the complete `if` statement along with the assocated `elif` and `else` clauses, it moves on to the next statement in the block containing the `if` statement. In this case, it is the main block where execution of the program starts and the next statement is the `print 'Done'` statement. After this, Python sees the ends of the program and simply finishes up.

Although this is a very simple program, I have been pointing out a lot of things that you should notice even in this simple program. All these are pretty straightforward (and surprisingly simple for those of you from C/C++ backgrounds) and requires you to become aware of all these initially, but after that, you will become comfortable with it and it'll feel 'natural' to you.

### Note for C/C++ Programmers

There is no `switch` statement in Python. You can use an `if..elif..else` statement to do the same thing (and in some cases, use a dictionary to do it quickly)

## The while statement

The `while` statement allows you to repeatedly execute a block of statements as long as a condition is true. A `while` statement is an example of what is called a *looping* statement. A `while` statement can have an optional `else` clause.

## Using the while statement

**Example 6.2. Using the while statement**

```
#!/usr/bin/python
# Filename: while.py

number = 23
running = True

while running:
        guess = int(raw_input('Enter an integer : '))

        if guess == number:
                print 'Congratulations, you guessed it.'
                running = False # this causes the while loop to stop
        elif guess < number:
                print 'No, it is a little higher than that.'
        else:
                print 'No, it is a little lower than that.'
else:
        print 'The while loop is over.'
        # Do anything else you want to do here

print 'Done'
```

## Output

```
$ python while.py
Enter an integer : 50
No, it is a little lower than that.
Enter an integer : 22
No, it is a little higher than that.
```

```
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over.
Done
```

### How It Works

In this program, we are still playing the guessing game, but the advantage is that the user is allowed to keep guessing until he guesses correctly - there is no need to repeatedly execute the program for each guess as we have done previously. This aptly demonstrates the use of the `while` statement.

We move the `raw_input` and `if` statements to inside the `while` loop and set the variable `running` to `True` before the while loop. First, we check if the variable `running` is `True` and then proceed to execute the corresponding *while-block*. After this block is executed, the condition is again checked which in this case is the `running` variable. If it is true, we execute the while-block again, else we continue to execute the optional else-block and then continue to the next statement.

The `else` block is executed when the `while` loop condition becomes `False` - this may even be the first time that the condition is checked. If there is an `else` clause for a `while` loop, it is always executed unless you have a `while` loop which loops forever without ever breaking out!

The `True` and `False` are called Boolean types and you can consider them to be equivalent to the value `1` and `0` respecitvely. It's important to use these where the condition or checking is important and not the actual value such as `1`.

The else-block is actually redundant since you can put those statements in the same block (as the `while` statement) after the `while` statement to get the same effect.

#### Note for C/C++ Programmers

Remember that you can have an `else` clause for the `while` loop.

## The for loop

The `for..in` statement is another looping statement which *iterates* over a sequence of objects i.e. go through each item in a sequence. We will see more about sequences in detail in later chapters. What you need to know right now is that a sequence is just an ordered collection of items.

## Using the for statement

**Example 6.3. Using the for statement**

```
#!/usr/bin/python
# Filename: for.py

for i in range(1, 5):
        print i
else:
        print 'The for loop is over'
```

### Output

```
$ python for.py
1
2
3
4
The for loop is over
```

### How It Works

In this program, we are printing a *sequence* of numbers. We generate this sequence of numbers using hte built-in `range` function.

What we do here is supply it two numbers and `range` returns a sequence of numbers starting from the first number and up to the second number. For example, `range(1,5)` gives the sequence `[1, 2, 3, 4]`. By default, `range` takes a step count of 1. If we supply a third number to `range`, then that becomes the step count. For example, `range(1,5,2)` gives `[1,3]`. Remember that the range extends *up to* the second number i.e. it does **not** include the second number.

The `for` loop then iterates over this range - `for i in range(1,5)` is equivalent to `for i in [1, 2, 3, 4]` which is like assigning each number (or object) in the sequence to i, one at a time, and then executing the block of statements for each value of `i`. In this case, we just print the value in the block of statements.

Remember that the `else` part is optional. When included, it is always executed once after the `for` loop is over unless a break statement is encountered.

Remember that the `for..in` loop works for any sequence. Here, we have a list of numbers generated by the built-in `range` function, but in general we can use any kind of sequence of any kind of objects! We will explore this idea in detail in later chapters.

#### Note for C/C++/Java/C# Programmers

The Python `for` loop is radically different from the C/C++ `for` loop. C# programmers will note that the `for` loop in Python is similar to the `foreach` loop in C#. Java programmers will note that the same is similar to `for (int i : IntArray)` in Java 1.5 .

In C/C++, if you want to write `for (int i = 0; i < 5; i++)`, then in Python you write just `for i in range(0,5)`. As you can see, the `for` loop is simpler, more expressive and less error prone in Python.

## The break statement

The `break` statement is used to *break* out of a loop statement i.e. stop the execution of a looping statement, even if the loop condition has not become `False` or the sequence of items has been completely iterated over.

An important note is that if you *break* out of a `for` or `while` loop, any corresponding loop `else` block is **not** executed.

### Using the break statement

**Example 6.4. Using the break statement**

```
#!/usr/bin/python
# Filename: break.py

while True:
        s = raw_input('Enter something : ')
        if s == 'quit':
                break
        print 'Length of the string is', len(s)
print 'Done'
```

## Output

```
$ python break.py
Enter something : Programming is fun
Length of the string is 18
Enter something : When the work is done
Length of the string is 21
Enter something : if you wanna make your work also fun:
Length of the string is 37
Enter something :       use Python!
Length of the string is 12
Enter something : quit
Done
```

## How It Works

In this program, we repeatedly take the user's input and print the length of each input each time. We are providing a special condition to stop the program by checking if the user input is `'quit'`. We stop the program by *break*ing out of the loop and reach the end of the program.

The length of the input string can be found out using the built-in `len` function.

Remember that the `break` statement can be used with the `for` loop as well.

## G2's Poetic Python

The input I have used here is a mini poem I have written called **G2's Poetic Python**:

```
Programming is fun
When the work is done
if you wanna make your work also fun:
        use Python!
```

## The continue statement

The `continue` statement is used to tell Python to skip the rest of the statements in the current loop block and to *continue* to the next iteration of the loop.

## Using the continue statement

**Example 6.5. Using the continue statement**

```
#!/usr/bin/python
# Filename: continue.py

while True:
        s = raw_input('Enter something : ')
        if s == 'quit':
                break
        if len(s) < 3:
                continue
        print 'Input is of sufficient length'
        # Do other kinds of processing here...
```

### Output

```
$ python continue.py
Enter something : a
Enter something : 12
Enter something : abc
Input is of sufficient length
Enter something : quit
```

### How It Works

In this program, we accept input from the user, but we process them only if they are at least 3 characters long. So, we use the built-in `len` function to get the length and if the length is less than 3, we skip the rest of the statements in the block by using the `continue` statement. Otherwise, the rest of the statements in the loop are executed and we can do any kind of processing we want to do here.

Note that the `continue` statement works with the `for` loop as well.

## Summary

We have seen how to use the three control flow statements - `if`, `while` and `for` along with their associated `break` and `continue` statements. These are some of the most often used parts of Python and hence, becoming comfortable with them is essential.

Next, we will see how to create and use functions.

# Chapter 7. Functions

## Introduction

Functions are reusable pieces of programs. They allow you to give a name to a block of statements and you can run that block using that name anywhere in your program and any number of times. This is known as *calling* the function. We have already used many built-in functions such as the `len` and `range`.

Functions are **def**ined using the `def` keyword. This is followed by an *identifier* name for the function followed by a pair of parentheses which may enclose some names of variables and the line ends with a colon. Next follows the block of statements that are part of this function. An example will show that this is actually very simple:

## Defining a Function

**Example 7.1. Defining a function**

```
#!/usr/bin/python
# Filename: function1.py

def sayHello():
	print 'Hello World!' # block belonging to the function
# End of function

sayHello() # call the function
```

### Output

```
$ python function1.py
Hello World!
```

### How It Works

We define a function called `sayHello` using the syntax as explained above. This function takes no parameters and hence there are no variables declared in the parentheses. Parameters to functions are just input to the function so that we can pass in different values to it and get back corresponding results.

## Function Parameters

A function can take parameters which are just values you supply to the function so that the function can *do* something utilising those values. These parameters are just like variables except that the values of these variables are defined when we call the function and are not assigned values within the function itself.

Parameters are specified within the pair of parentheses in the function definition, separated by commas. When we call the function, we supply the values in the same way. Note the terminology used - the names given in the function definition are called *parameters* whereas the values you supply in the function call are called *arguments*.

## Using Function Parameters

**Example 7.2. Using Function Parameters**

```
#!/usr/bin/python
# Filename: func_param.py

def printMax(a, b):
        if a > b:
                print a, 'is maximum'
        else:
                print b, 'is maximum'

printMax(3, 4) # directly give literal values

x = 5
y = 7

printMax(x, y) # give variables as arguments
```

### Output

```
$ python func_param.py
4 is maximum
7 is maximum
```

### How It Works

Here, we define a function called `printMax` where we take two parameters called `a` and `b`. We find out the greater number using a simple `if..else` statement and then print the bigger number.

In the first usage of `printMax`, we directly supply the numbers i.e. arguments. In the second usage, we call the function using variables. `printMax(x, y)` causes value of argument `x` to be assigned to parameter `a` and the value of argument `y` assigned to parameter `b`. The printMax function works the same in both the cases.

## Local Variables

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function i.e. variable names are *local* to the function. This is called the *scope* of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

## Using Local Variables

**Example 7.3. Using Local Variables**

```
#!/usr/bin/python
# Filename: func_local.py

def func(x):
        print 'x is', x
        x = 2
        print 'Changed local x to', x

x = 50
func(x)
print 'x is still', x
```

### Output

```
$ python func_local.py
x is 50
Changed local x to 2
x is still 50
```

### How It Works

In the function, the first time that we use the *value* of the name `x`, Python uses the value of the parameter declared in the function.

Next, we assign the value `2` to `x`. The name `x` is local to our function. So, when we change the value of `x` in the function, the `x` defined in the main block remains unaffected.

In the last `print` statement, we confirm that the value of `x` in the main block is actually unaffected.

## Using the global statement

If you want to assign a value to a name defined outside the function, then you have to tell Python that the name is not local, but it is *global*. We do this using the `global` statement. It is impossible to assign a value to a variable defined outside a function without the `global` statement.

You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the `global` statement makes it amply clear that the variable is defined in an outer block.

**Example 7.4. Using the global statement**

```
#!/usr/bin/python
# Filename: func_global.py

def func():
        global x

        print 'x is', x
        x = 2
        print 'Changed global x to', x

x = 50
func()
print 'Value of x is', x
```

### Output

```
$ python func_global.py
x is 50
Changed global x to 2
Value of x is 2
```

### How It Works

The `global` statement is used to decare that `x` is a global variable - hence, when we assign a value to `x` inside the function, that change is reflected when we use the value of `x` in the main block.

You can specify more than one global variable using the same `global` statement. For example, `global x, y, z`.

## Default Argument Values

For some functions, you may want to make some of its parameters as *optional* and use default values if the user does not want to provide values for such parameters. This is done with the help of default argument values. You can specify default argument values for parameters by following the parameter name in the function definition with the assignment operator (=) followed by the default value.

Note that the default argument value should be a constant. More precisely, the default argument value should be immutable - this is explained in detail in later chapters. For now, just remember this.

### Using Default Argument Values

**Example 7.5. Using Default Argument Values**

```
#!/usr/bin/python
# Filename: func_default.py

def say(message, times = 1):
        print message * times

say('Hello')
say('World', 5)
```

### Output

```
$ python func_default.py
Hello
WorldWorldWorldWorldWorld
```

### How It Works

The function named `say` is used to print a string as many times as want. If we don't supply a value, then by default, the string is printed just once. We achieve this by specifying a default argument value of `1` to the parameter `times`.

In the first usage of `say`, we supply only the string and it prints the string once. In the second usage of `say`, we supply both the string and an argument `5` stating that we want to *say* the string message 5 times.

#### Important

Only those parameters which are at the end of the parameter list can be given default argument values i.e. you cannot have a parameter with a default argument value before a parameter without a default argument value in the order of parameters declared in the function parameter list.

This is because the values are assigned to the parameters by position. For example, `def func(a, b=5)` is valid, but `def func(a=5, b)` is *not valid*.

## Keyword Arguments

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called *keyword arguments* - we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

There are two *advantages* - one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters which we want, provided that the other parameters have default argument values.

## Using Keyword Arguments

**Example 7.6. Using Keyword Arguments**

```
#!/usr/bin/python
# Filename: func_key.py

def func(a, b=5, c=10):
        print 'a is', a, 'and b is', b, 'and c is', c

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

### Output

```
$ python func_key.py
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

### How It Works

The function named `func` has one parameter without default argument values, followed by two parameters with default argument values.

In the first usage, `func(3, 7)`, the parameter `a` gets the value `3`, the parameter `b` gets the value `5` and `c` gets the default value of `10`.

In the second usage `func(25, c=24)`, the variable `a` gets the value of 25 due to the position of the argument. Then, the parameter `c` gets the value of `24` due to naming i.e. keyword arguments. The variable `b` gets the default value of `5`.

In the third usage `func(c=50, a=100)`, we use keyword arguments completely to specify the values. Notice, that we are specifying value for parameter `c` before that for `a` even though `a` is defined before `c` in the function definition.

## The return statement

The `return` statement is used to *return* from a function i.e. break out of the function. We can optionally *return a value* from the function as well.

## Using the literal statement

**Example 7.7. Using the literal statement**

```
#!/usr/bin/python
# Filename: func_return.py

def maximum(x, y):
        if x > y:
                return x
        else:
                return y

print maximum(2, 3)
```

### Output

```
$ python func_return.py
3
```

### How It Works

The `maximum` function returns the maximum of the parameters, in this case the numbers supplied to the function. It uses a simple `if..else` statement to find the greater value and then *returns* that value.

Note that a `return` statement without a value is equivalent to `return None`. `None` is a special type in Python that represents nothingness. For example, it is used to indicate that a variable has no value if it has a value of `None`.

Every function implicitly contains a `return None` statement at the end unless you have written your own `return` statement. You can see this by running `print someFunction()` where the function `someFunction` does not use the `return` statement such as:

```
def someFunction():
        pass
```

The `pass` statement is used in Python to indicate an empty block of statements.

## DocStrings

Python has a nifty feature called *documentation strings* which is usually referred to by its shorter name *docstrings*. DocStrings are an important tool that you should make use of since it helps to document the program better and makes it more easy to understand. Amazingly, we can even get back the docstring from, say a function, when the program is actually running!

## Using DocStrings

**Example 7.8. Using DocStrings**

```
#!/usr/bin/python
# Filename: func_doc.py

def printMax(x, y):
        '''Prints the maximum of two numbers.

        The two values must be integers.'''
        x = int(x) # convert to integers, if possible
        y = int(y)

        if x > y:
                print x, 'is maximum'
        else:
                print y, 'is maximum'

printMax(3, 5)
print printMax.__doc__
```

## Output

```
$ python func_doc.py
5 is maximum
Prints the maximum of two numbers.

        The two values must be integers.
```

## How It Works

A string on the first logical line of a function is the *docstring* for that function. Note that DocStrings also apply to modules and classes which we will learn about in the respective chapters.

The convention followed for a docstring is a multi-line string where the first line starts with a capital letter and ends with a dot. Then the second line is blank followed by any detailed explanation starting from the third line. You are *strongly advised* to follow this convention for all your docstrings for all your non-trivial functions.

We can access the docstring of the `printMax` function using the `__doc__` (notice the double underscores) attribute (name belonging to) of the function. Just remember that Python treats *everything* as an object and this includes functions. We'll learn more about objects in the chapter on classes.

If you have used the `help()` in Python, then you have already seen the usage of docstrings! What it does is just fetch the `__doc__` attribute of that function and displays it in a neat manner for you. You can try it out on the function above - just include `help(printMax)` in your program. Remember to press **q** to exit the `help`.

Automated tools can retrieve the documentation from your program in this manner. Therefore, I *strongly recommend* that you use docstrings for any non-trivial function that you write. The **pydoc** command that comes with your Python distribution works similarly to `help()` using docstrings.

## Summary

We have seen so many aspects of functions but note that we still haven't covered all aspects of it. However, we have already covered most of what you'll use regarding Python functions on an everyday basis.

Next, we will see how to use as well as create Python modules.

# Chapter 8. Modules

## Introduction

You have seen how you can reuse code in your program by defining functions once. What if you wanted to reuse a number of functions in other programs that you write? As you might have guessed, the answer is modules. A module is basically a file containing all your functions and variables that you have defined. To reuse the module in other programs, the filename of the module **must** have a `.py` extension.

A module can be *imported* by another program to make use of its functionality. This is how we can use the Python standard library as well. First, we will see how to use the standard library modules.

## Using the sys module

**Example 8.1. Using the sys module**

```
#!/usr/bin/python
# Filename: using_sys.py

import sys

print 'The command line arguments are:'
for i in sys.argv:
        print i

print '\n\nThe PYTHONPATH is', sys.path, '\n'
```

### Output

```
$ python using_sys.py we are arguments
The command line arguments are:
using_sys.py
we
are
arguments


The PYTHONPATH is ['/home/swaroop/byte/code', '/usr/lib/python23.zip',
'/usr/lib/python2.3', '/usr/lib/python2.3/plat-linux2',
'/usr/lib/python2.3/lib-tk', '/usr/lib/python2.3/lib-dynload',
'/usr/lib/python2.3/site-packages', '/usr/lib/python2.3/site-packages/gtk-2.0']
```

### How It Works

First, we *import* the `sys` module using the `import` statement. Basically, this translates to us telling Python that we want to use this module. The `sys` module contains functionality related to the Python interpreter and its environment.

When Python executes the `import sys` statement, it looks for the `sys.py` module in one of the directores listed in its `sys.path` variable. If the file is found, then the statements in the main block of that module is run and then the module is made *available* for you to use. Note that the initialization is done only the *first* time that we import a module. Also, 'sys' is short for 'system'.

The `argv` variable in the `sys` module is referred to using the dotted notation - `sys.argv` - one of the advantages of this approach is that the name does not clash with any `argv` variable used in your program. Also, it indicates clearly that this name is part of the `sys` module.

The `sys.argv` variable is a *list* of strings (lists are explained in detail in later sections). Specifically, the `sys.argv` contains the list of *command line arguments* i.e. the arguments passed to your program using the command line.

If you are using an IDE to write and run these programs, look for a way to specify command line arguments to the program in the menus.

Here, when we execute `python using_sys.py we are arguments`, we run the module `using_sys.py` with the **python** command and the other things that follow are arguments passed to the program. Python stores it in the `sys.argv` variable for us.

Remember, the name of the script running is always the first argument in the `sys.argv` list. So, in this case we will have `'using_sys.py'` as `sys.argv[0]`, `'we'` as `sys.argv[1]`, `'are'` as `sys.argv[2]` and `'arguments'` as `sys.argv[3]` . Notice that Python starts counting from 0 and not 1.

The `sys.path` contains the list of directory names where modules are imported from. Observe that the first string in `sys.path` is empty - this empty string indicates that the current directory is also part of the `sys.path` which is same as the `PYTHONPATH` environment variable. This means that you can directly import modules located in the current directory. Otherwise, you will have to place your module in one of the directories listed in `sys.path` .

## Byte-compiled .pyc files

Importing a module is a relatively costly affair, so Python does some tricks to make it faster. One way is to create *byte-compiled* files with the extension `.pyc` which is related to the intermediate form that Python transforms the program into (remember the intro section on how Python works ?). This `.pyc` file is useful when you import the module the next time from a different program - it will be much faster since part of the processing required in importing a module is already done. Also, these byte-compiled files are platform-independent. So, now you know what those `.pyc` files really are.

## The from..import statement

If you want to directly import the `argv` variable into your program (to avoid typing the `sys.` everytime for it), then you can use the `from sys import argv` statement. If you want to import all the names used in the `sys` module, then you can use the `from sys import *` statement. This works for any module. In general, avoid using the `from..import` statement and use the `import` statement instead since your program will be much more readable and will avoid any name clashes that way.

## A module's __name__

Every module has a name and statements in a module can find out the name of its module. This is especially handy in one particular situation - As mentioned previously, when a module is imported for the first time, the main block in that module is run. What if we want to run the block only if the program was used by itself and not when it was imported from another module? This can be achieved using the `__name__` attribute of the module.

## Using a module's __name__

**Example 8.2. Using a module's __name__**

```
#!/usr/bin/python
# Filename: using_name.py

if __name__ == '__main__':
        print 'This program is being run by itself'
else:
        print 'I am being imported from another module'
```

### Output

```
$ python using_name.py
This program is being run by itself

$ python
>>> import using_name
I am being imported from another module
>>>
```

### How It Works

Every Python module has it's `__name__` defined and if this is `'__main__'`, it implies that the module is being run standalone by the user and we can do corresponding appropriate actions.

## Making your own Modules

Creating your own modules is easy, you've been doing it all along! Every Python program is also a module. You just have to make sure it has a `.py` extension. The following example should make it clear.

## Creating your own Modules

**Example 8.3. How to create your own module**

```
#!/usr/bin/python
# Filename: mymodule.py

def sayhi():
        print 'Hi, this is mymodule speaking.'

version = '0.1'

# End of mymodule.py
```

The above was a sample *module*. As you can see, there is nothing particularly special about compared to our usual Python program. We will next see how to use this module in our other Python programs.

Remember that the module should be placed in the same directory as the program that we import it in, or the module should be in one of the directories listed in `sys.path` .

```
#!/usr/bin/python
# Filename: mymodule_demo.py

import mymodule

mymodule.sayhi()
print 'Version', mymodule.version
```

### Output

```
$ python mymodule_demo.py
Hi, this is mymodule speaking.
Version 0.1
```

### How It Works

Notice that we use the same dotted notation to access members of the module. Python makes good reuse of the same notation to give the distinctive 'Pythonic' feel to it so that we don't have to keep learning new ways to do things.

## from..import

Here is a version utilising the `from..import` syntax.

```
#!/usr/bin/python
# Filename: mymodule_demo2.py

from mymodule import sayhi, version
# Alternative:
```

```
# from mymodule import *

sayhi()
print 'Version', version
```

The output of `mymodule_demo2.py` is same as the output of `mymodule_demo.py`.

## The dir() function

You can use the built-in `dir` function to list the identifiers that a module defines. The identifiers are the functions, classes and variables defined in that module.

When you supply a module name to the `dir()` function, it returns the list of the names defined in that module. When no argument is applied to it, it returns the list of names defined in the current module.

## Using the dir function

**Example 8.4. Using the dir function**

```
$ python
>>> import sys
>>> dir(sys) # get list of attributes for sys module
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
'__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
'copyright', 'displayhook', 'exc_clear', 'exc_info', 'exc_type',
'excepthook', 'exec_prefix', 'executable', 'exit', 'getcheckinterval',
'getdefaultencoding', 'getdlopenflags', 'getfilesystemencoding',
'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
'meta_path','modules', 'path', 'path_hooks', 'path_importer_cache',
'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
'version', 'version_info', 'warnoptions']
>>> dir() # get list of attributes for current module
['__builtins__', '__doc__', '__name__', 'sys']
>>>
>>> a = 5 # create a new variable 'a'
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'sys']
>>>
>>> del a # delete/remove a name
>>>
>>> dir()
['__builtins__', '__doc__', '__name__', 'sys']
>>>
```

### How It Works

First, we see the usage of `dir` on the imported `sys` module. We can see the huge list of attributes that it contains.

Next, we use the `dir` function without passing parameters to it - by default, it returns the list of attributes for the current module. Notice that the list of imported modules is also part of this list.

In order to observe the `dir` in action, we define a new variable `a` and assign it a value and then check `dir` and we observe that there is an additional value in the list of the same name. We remove the variable/attribute of the current module using the `del` statement and the change is reflected again in the output of the `dir` function.

A note on `del` - this statement is used to *delete* a variable/name and after the statement has run, in this case `del a`, you can no longer access the variable `a` - it is as if it never existed before at all.

## Summary

Modules are useful because they provide services and functionality that you can reuse in other programs. The standard library that comes with Python is an example of such a set of modules. We have seen how to use these modules and create our own modules as well.

Next, we will learn about some interesting concepts called data structures.

# Chapter 9. Data Structures

## Introduction

Data structures are basically just that - they are *structures* which can hold some *data* together. In other words, they are used to store a collection of related data.

There are three built-in data structures in Python - list, tuple and dictionary. We will see how to use each of them and how they make life easier.

## List

A `list` is a data structure that holds an ordered collection of items i.e. you can store a *sequence* of items in a list. This is easy to imagine if you can think of a shopping list where you have a list of items to buy, except that you probbly have each item on a separate line in your shopping list whereas in Python you put commas in between them.

The list of items should be enclosed in square brackets so that Python understands that you are specifying a list. Once you have created a list, you can add, remove or search for items in the list. Since, we can add and remove items, we say that a list is a *mutable* data type i.e. this type can be altered.

### Quick introduction to Objects and Classes

Although, I've been generally delaying the discussion of objects and classes till now, a little explanation is needed right now so that you can understand lists better. We will still explore this topic in detail in its own chapter.

A list is an example of usage of objects and classes. When you use a variable `i` and assign a value to it, say integer `5` to it, you can think of it as creating an **object** (instance) `i` of **class** (type) `int`. In fact, you can see `help(int)` to understand this better.

A class can also have **methods** i.e. functions defined for use with respect to that class only. You can use these pieces of functionality only when you have an object of that class. For example, Python provides an `append` method for the `list` class which allows you to add an item to the end of the list. For example, `mylist.append('an item')` will add that string to the list `mylist`. Note the use of dotted notation for accessing methods of the objects.

A class can also have **fields** which are nothing but variables defined for use with respect to that class only. You can use these variables/names only when you have an object of that class. Fields are also accessed by the dotted notation, for example, `mylist.field` .

### Using Lists

**Example 9.1. Using lists**

```
#!/usr/bin/python
# Filename: using_list.py

# This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']
```

```
print 'I have', len(shoplist), 'items to purchase.'

print 'These items are:', # Notice the comma at end of the line
for item in shoplist:
        print item,

print '\nI also have to buy rice.'
shoplist.append('rice')
print 'My shopping list is now', shoplist

print 'I will sort my list now'
shoplist.sort()
print 'Sorted shopping list is', shoplist

print 'The first item I will buy is', shoplist[0]
olditem = shoplist[0]
del shoplist[0]
print 'I bought the', olditem
print 'My shopping list is now', shoplist
```

## Output

```
$ python using_list.py
I have 4 items to purchase.
These items are: apple mango carrot banana
I also have to buy rice.
My shopping list is now ['apple', 'mango', 'carrot', 'banana', 'rice']
I will sort my list now
Sorted shopping list is ['apple', 'banana', 'carrot', 'mango', 'rice']
The first item I will buy is apple
I bought the apple
My shopping list is now ['banana', 'carrot', 'mango', 'rice']
```

## How It Works

The variable `shoplist` is a shopping list for someone who is going to the market. In `shoplist`, we only store strings of the names of the items to buy but remember you can add *any kind of object* to a list including numbers and even other lists.

We have also used the `for..in` loop to iterate through the items of the list. By now, you must have realised that a list is also a sequence. The speciality of sequences will be discussed in a later section

Notice that we use a *comma* at the end of the `print` statement to suppress the automatic printing of a line break after every `print` statement. This is a bit of an ugly way of doing it, but it is simple and gets the job done.

Next, we add an item to the list using the `append` method of the list object, as already discussed before. Then, we check that the item has been indeed added to the list by printing the contents of the list by simply passing the list to the `print` statement which prints it in a neat manner for us.

Then, we sort the list by using the `sort` method of the list. Understand that this method affects the list itself and does not return a modified list - this is different from the way strings work. This is what we mean by saying that lists are *mutable* and that strings are *immutable*.

Next, when we finish buying an item in the market, we want to remove it from the list. We achieve this by using the `del` statement. Here, we mention which item of the list we want to remove and the `del` statement removes it fromt he list for us. We specify that we want to remove the first item from the list and hence we use `del shoplist[0]` (remember that Python starts counting from 0).

If you want to know all the methods defined by the list object, see `help(list)` for complete details.

## Tuple

Tuples are just like lists except that they are **immutable** like strings i.e. you cannot modify tuples. Tuples are defined by specifying items separated by commas within a pair of parentheses. Tuples are usually used in cases where a statement or a user-defined function can safely assume that the collection of values i.e. the tuple of values used will not change.

## Using Tuples

**Example 9.2. Using Tuples**

```
#!/usr/bin/python
# Filename: using_tuple.py

zoo = ('wolf', 'elephant', 'penguin')
print 'Number of animals in the zoo is', len(zoo)

new_zoo = ('monkey', 'dolphin', zoo)
print 'Number of animals in the new zoo is', len(new_zoo)
print 'All animals in new zoo are', new_zoo
print 'Animals brought from old zoo are', new_zoo[2]
print 'Last animal brought from old zoo is', new_zoo[2][2]
```

### Output

```
$ python using_tuple.py
Number of animals in the zoo is 3
Number of animals in the new zoo is 3
All animals in new zoo are ('monkey', 'dolphin', ('wolf', 'elephant', 'penguin'))
Animals brought from old zoo are ('wolf', 'elephant', 'penguin')
Last animal brought from old zoo is penguin
```

### How It Works

The variable `zoo` refers to a tuple of items. We see that the `len` function can be used to get the length of the tuple. This also indicates that a tuple is a sequence as well.

We are now shifting these animals to a new zoo since the old zoo is being closed. Therefore, the `new_zoo` tuple contains some animals which are already there along with the animals brought over from the old zoo. Back to reality, note that a tuple within a tuple does not lose its identity.

We can access the items in the tuple by specifying the item's position within a pair of square brackets just like we did for lists. This is called the *indexing* operator. We access the third item in `new_zoo` by specifying `new_zoo[2]` and we access the third item in the third item in the `new_zoo` tuple by specifying `new_zoo[2][2]`. This is pretty simple once you've understood the idiom.

**Tuple with 0 or 1 items.** An empty tuple is constructed by an empty pair of parentheses such as `myempty = ()`. However, a tuple with a single item is not so simple. You have to specify it using a comma following the first (and only) item so that Python can differentiate between a tuple and a pair of parentheses surrounding the object in an expression i.e. you have to specify `singleton = (2 , )` if you mean you want a tuple containing the item `2`.

### Note for Perl programmers

A list within a list does not lose its identity i.e. lists are not flattened as in Perl. The same applies to a tuple within a tuple, or a tuple within a list, or a list within a tuple, etc. As far as Python is concerned, they are just objects stored using another object, that's all.

## Tuples and the print statement

One of the most common usage of tuples is with the print statement. Here is an example:

**Example 9.3. Output using tuples**

```
#!/usr/bin/python
# Filename: print_tuple.py

age = 22
name = 'Swaroop'

print '%s is %d years old' % (name, age)
print 'Why is %s playing with that python?' % name
```

### Output

```
$ python print_tuple.py
Swaroop is 22 years old
Why is Swaroop playing with that python?
```

### How It Works

The `print` statement can take a string using certain specifications followed by the `%` symbol followed by a tuple of items matching the specification. The specifications are used to format the output in a certain way. The specification can be like `%s` for strings and `%d` for integers. The tuple must have items corresponding to these specifications in the same order.

Observe the first usage where we use `%s` first and this corresponds to the variable `name` which is the first item in the tuple and the second specification is `%d` corresponding to `age` which is the second item in the tuple.

What Python does here is that it converts each item in the tuple into a string and substitutes that string value into the place of the specification. Therefore the `%s` is replaced by the value of the variable `name` and so on.

This usage of the `print` statement makes writing output extremely easy and avoids lot of string manipulation to achieve the same. It also avoids using commas everywhere as we have done till now.

Most of the time, you can just use the `%s` specification and let Python take care of the rest for you. This works even for numbers. However, you may want to give the correct specifications since this adds one level of checking that your program is correct.

In the second `print` statement, we are using a single specification followed by the `%` symbol followed by a single item - there are no pair of parentheses. This works only in the case where there is a single specification in the string.

## Dictionary

A dictionary is like an address-book where you can find the address or contact details of a person by knowing only his/her name i.e. we associate **keys** (name) with **values** (details). Note that the key must be unique just like you cannot find out the correct information if you have two persons with the exact same name.

Note that you can use only immutable objects (like strings) for the keys of a dictionary but you can use either immutable or mutable objects for the values of the dictionary. This basically translates to say that you should use only simple objects for keys.

Pairs of keys and valus are specified in a dictionary by using the notation `d = {key1 : value1, key2 : value2 }`. Notice that they key/value pairs are separated by a colon and the pairs are separated themselves by commas and all this is enclosed in a pair of curly brackets.

Remember that key/value pairs in a dictionary are not ordered in any manner. If you want a particular order, then you will have to sort them yourself before using it.

The dictionaries that you will be using are instances/objects of the `dict` class.

## Using Dictionaries

### Example 9.4. Using dictionaries

```
#!/usr/bin/python
# Filename: using_dict.py

# 'ab' is short for 'a'ddress'b'ook

ab = {          'Swaroop'   : 'swaroopch@byteofpython.info',
                 'Larry'     : 'larry@wall.org',
                 'Matsumoto' : 'matz@ruby-lang.org',
                 'Spammer'   : 'spammer@hotmail.com'
```

```
        }

print "Swaroop's address is %s" % ab['Swaroop']

# Adding a key/value pair
ab['Guido'] = 'guido@python.org'

# Deleting a key/value pair
del ab['Spammer']

print '\nThere are %d contacts in the address-book\n' % len(ab)

for name, address in ab.items():
        print 'Contact %s at %s' % (name, address)

if 'Guido' in ab: # OR ab.has_key('Guido')
        print "\nGuido's address is %s" % ab['Guido']
```

## Output

```
$ python using_dict.py
Swaroop's address is swaroopch@byteofpython.info

There are 4 contacts in the address-book

Contact Swaroop at swaroopch@byteofpython.info
Contact Matsumoto at matz@ruby-lang.org
Contact Larry at larry@wall.org
Contact Guido at guido@python.org

Guido's address is guido@python.org
```

## How It Works

We create the dictionary `ab` using the notation already discussed. We then access key/value pairs by specifying the key using the indexing operator as discussed in the context of lists and tuples. Observe that the syntax is very simple for dictionaries as well.

We can add new key/value pairs by simply using the indexing operator to access a key and assign that value, as we have done for Guido in the above case.

We can delete key/value pairs using our old friend - the `del` statement. We simply specify the dictionary and the indexing operator for the key to be removed and pass it to the `del` statement. There is no need to know the value corresponding to the key for this operation.

Next, we access each key/value pair of the dictionary using the `items` method of the dictionary which returns a list of tuples where each tuple contains a pair of items - the key followed by the value. We retrieve this pair and assign it to the variables `name` and `address` correspondingly for each pair using the `for..in` loop and then print these values in the for-block.

We can check if a key/value pair exists using the `in` operator or even the `has_key` method of the `dict` class. You can see the documentation for the complete list of methods of the `dict` class using `help(dict)`.

**Keyword Arguments and Dictionaries.** On a different note, if you have used keyword arguments in your functions, you have already used dictionaries! Just think about it - the key/value pair is specified by you in the parameter list of the function definition and when you access variables within your function, it is just a key access of a dictionary (which is called the *symbol table* in compiler design terminology).

## Sequences

Lists, tuples and strings are examples of sequences, but what are sequences and what is so special about them? Two of the main features of a sequence is the **indexing** operation which allows us to fetch a particular item in the sequence directly and the **slicing** operation which allows us to retrieve a slice of the sequence i.e. a part of the sequence.

## Using Sequences

### Example 9.5. Using Sequences

```
#!/usr/bin/python
# Filename: seq.py

shoplist = ['apple', 'mango', 'carrot', 'banana']

# Indexing or 'Subscription' operation
print 'Item 0 is', shoplist[0]
print 'Item 1 is', shoplist[1]
print 'Item 2 is', shoplist[2]
print 'Item 3 is', shoplist[3]
print 'Item -1 is', shoplist[-1]
print 'Item -2 is', shoplist[-2]

# Slicing on a list
print 'Item 1 to 3 is', shoplist[1:3]
print 'Item 2 to end is', shoplist[2:]
print 'Item 1 to -1 is', shoplist[1:-1]
print 'Item start to end is', shoplist[:]

# Slicing on a string
name = 'swaroop'
print 'characters 1 to 3 is', name[1:3]
print 'characters 2 to end is', name[2:]
print 'characters 1 to -1 is', name[1:-1]
print 'characters start to end is', name[:]
```

## Output

```
$ python seq.py
Item 0 is apple
Item 1 is mango
Item 2 is carrot
Item 3 is banana
Item -1 is banana
Item -2 is carrot
Item 1 to 3 is ['mango', 'carrot']
Item 2 to end is ['carrot', 'banana']
Item 1 to -1 is ['mango', 'carrot']
Item start to end is ['apple', 'mango', 'carrot', 'banana']
characters 1 to 3 is wa
characters 2 to end is aroop
characters 1 to -1 is waroo
characters start to end is swaroop
```

### How It Works

First, we see how to use indexes to get individual items of a sequence. This is also referred to as the subscription operation. Whenever you specify a number to a sequence within square brackets as shown above, Python will fetch you the item corresponding to that position in the sequence. Remember that Python starts counting numbers from 0. Hence, `shoplist[0]` fetches the first item and `shoplist[3]` fetches the fourth item in the `shoplist` sequence.

The index can also be a negative number, in which case, the position is calculated from the end of the sequence. Therefore, `shoplist[-1]` refers to the last item in the sequence and `shoplist[-2]` fetches the second last item in the sequence.

The slicing operation is used by specifying the name of the sequence followed by an optional pair of numbers separated by a colon within square brackets. Note that this is very very similar to the indexing operation you have been using til lnow. Remember the numbers are optional but the colon isn't.

The first number (before the colon) in the slicing operation refers to the position from where the slice starts and the second number (after the colon) indicates where the slice will stop at. If the first number is not specified, Python will start at the beginning of the sequence. If the second number is left out, Python will stop at the end of the sequence. Note that the slice returned *starts* at the start position and will end just before the *end* position i.e. the start position is included but the end position is excluded from the sequence slice.

Thus, `shoplist[1:3]` returns a slice of the sequence starting at position 1, includes position 2 but stops at position 3 and therefore a *slice* of two items is returned. Similarly, `shoplist[:]` returns a copy of the whole sequence.

You can also do slicing with negative positions. Negative numbers are used for positions from the end of the sequence. For example, `shoplist[:-1]` will return a slice of the sequence which excludes the last item of the sequence but contains everything else.

Try various combinations of such slice specifications using the Python interpreter interactively i.e. the prompt so that you can see the results immediately. The great thing about sequences is that you can access tuples, lists and strings all in the same way!

## References

When you create an object and assign it to a variable, the variable only *refers* to the object and does not represent the object itself! That is, the variable name points to that part of your computer's memory where the object is stored. This is called as **binding** of the name to the object.

Generally, you don't need to be worried about this, but there is a subtle effect due to references which you need to be aware of. This is demonstrated by the following example.

## Objects and References

**Example 9.6. Objects and References**

```
#!/usr/bin/python
# Filename: reference.py

print 'Simple Assignment'
shoplist = ['apple', 'mango', 'carrot', 'banana']
mylist = shoplist # mylist is just another name pointing to the same object!

del shoplist[0] # I purchased the first item, so I remove it from the list

print 'shoplist is', shoplist
print 'mylist is', mylist
# notice that both shoplist and mylist both print the same list without
# the 'apple' confirming that they point to the same object

print 'Copy by making a full slice'
mylist = shoplist[:] # make a copy by doing a full slice
del mylist[0] # remove first item

print 'shoplist is', shoplist
print 'mylist is', mylist
# notice that now the two lists are different
```

### Output

```
$ python reference.py
Simple Assignment
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
Copy by making a full slice
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

### How It Works

Most of the explanation is available in the comments itself. What you need to remember is that if you want to make a copy of a list or such kinds of sequences or complex objects (not simple *objects* such as integers), then you have to use the slicing operation to make a copy. If you just assign the variable name to another name, both of them will *refer* to the same object and this could lead to all sorts of trouble if you are not careful.

#### Note for Perl programmers

Remember that an assignment statement for lists does **not** create a copy. You have to use slicing operation to make a copy of the sequence.

## More about Strings

We have already discussed strings in detail earlier. What more can there be to know? Well, did you know that strings are also objects and have methods which do everything from checking part of a string to stripping spaces!

The strings that you use in program are all objects of the class `str`. Some useful methods of this class are demonstrated in the next example. For a complete list of such methods, see `help(str)`.

## String Methods

**Example 9.7. String Methods**

```
#!/usr/bin/python
# Filename: str_methods.py

name = 'Swaroop' # This is a string object

if name.startswith('Swa'):
        print 'Yes, the string starts with "Swa"'

if 'a' in name:
        print 'Yes, it contains the string "a"'

if name.find('war') != -1:
        print 'Yes, it contains the string "war"'

delimiter = '_*_'
mylist = ['Brazil', 'Russia', 'India', 'China']
print delimiter.join(mylist)
```

### Output

```
$ python str_methods.py
Yes, the string starts with "Swa"
Yes, it contains the string "a"
Yes, it contains the string "war"
Brazil_*_Russia_*_India_*_China
```

### How It Works

Here, we see a lot of the string methods in action. The `startswith` method is used to find out whether the string starts with the given string. The `in` operator is used to check if a given string is a part of the string.

The `find` method is used to do find the position of the given string in the string or returns -1 if it is not successful to find the substring. The `str` class also has a neat method to `join` the items of a sequence with the string acting as a delimiter between each item of the sequence and returns a bigger string generated from this.

## Summary

We have explored the various built-in data structures of Python in detail. These data structures will be essential for writing programs of reasonable size.

Now that we have a lot of the basics of Python in place, we will next see how to design and write a real-world Python program.

# Chapter 10. Problem Solving - Writing a Python Script

We have explored various parts of the Python language and now we will take a look at how all these parts fit together, by designing and writing a program which *does* something useful.

## The Problem

The problem is '*I want a program which creates a backup of all my important files*'.

Although, this is a simple problem, there is not enough information for us to get started with the solution. A little more **analysis** is required. For example, how do we specify which files are to be backed up? Where is the backup stored? How are they stored in the backup?

After analyzing the problem properly, we **design** our program. We make a list of things about how our program should work. In this case, I have created the following list on how *I* want it to work. If you do the design, you may not come up with the same kind of problem - every person has their own way of doing things, this is ok.

1. The files and directories to be backed up are specified in a list.
2. The backup must be stored in a main backup directory.
3. The files are backed up into a zip file.
4. The name of the zip archive is the current date and time.
5. We use the standard **zip** command available by default in any standard Linux/Unix distribution. Windows users can use the Info-Zip program. Note that you can use any archiving command you want as long as it has a command line interface so that we can pass arguments to it from our script.

## The Solution

As the design of our program is now stable, we can write the code which is an **implementation** of our solution.

### First Version

**Example 10.1. Backup Script - The First Version**

```
#!/usr/bin/python
# Filename: backup_ver1.py

import os
import time

# 1. The files and directories to be backed up are specified in a list.
source = ['/home/swaroop/byte', '/home/swaroop/bin']
```

```
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or somethin

# 2. The backup must be stored in a main backup directory
target_dir = '/mnt/e/backup/' # Remember to change this to what you will be using

# 3. The files are backed up into a zip file.
# 4. The name of the zip archive is the current date and time
target = target_dir + time.strftime('%Y%m%d%H%M%S') + '.zip'

# 5. We use the zip command (in Unix/Linux) to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))

# Run the backup
if os.system(zip_command) == 0:
        print 'Successful backup to', target
else:
        print 'Backup FAILED'
```

## Output

```
$ python backup_ver1.py
Successful backup to /mnt/e/backup/20041208073244.zip
```

Now, we are in the **testing** phase where we test that our program works properly. If it doesn't behave as expected, then we have to **debug** our program i.e. remove the *bugs* (errors) from the program.

## How It Works

You will notice how we have converted our *design* into *code* in a step-by-step manner.

We make use of the `os` and `time` modules and so we import them. Then, we specify the files and directories to be backed up in the `source` list. The target directory is where store all the backup files and this is specified in the `target_dir` variable. The name of the zip archive that we are going to create is the current date and time which we fetch using the `time.strftime()` function. It will also have the `.zip` extension and will be stored in the `target_dir` directory.

The `time.strftime()` function takes a specification such as the one we have used in the above program. The `%Y` specification will be replaced by the year without the cetury. The `%m` specification will be replaced by the month as a decimal number between `01` and `12` and so on. The complete list of such specifications can be found in the [Python Reference Manual] that comes with your Python distribution. Notice that this is similar to (but not same as) the specification used in `print` statement (using the `%` followed by tuple).

We create the name of the target zip file using the addition operator which *concatenates* the strings i.e. it joins the two strings together and returns a new one. Then, we create a string `zip_command` which contains the command that we are going to execute. You can check if this command works by running it on the shell (Linux terminal or DOS prompt).

The **zip** command that we are using has some options and parameters passed. The `-q` option is used to indicate that the zip command should work **q**uietly. The `-r` option specifies that the zip command should work **r**ecursively for directories i.e. it should include subdirectories and files within the subdirectories as well. The two options are combined and specified in a shorter way as `-qr`. The options are followed by the name of the zip archive to create followed by the list of files and directories to backup. We convert the `source` list into a string using the `join` method of strings which we have already seen how to use.

Then, we finally *run* the command using the `os.system` function which runs the command as if it was run from the *system* i.e. in the shell - it returns `0` if the command was successfully, else it returns an error number.

Depending on the outcome of the command, we print the appropriate message that the backup has failed or succeeded and that's it, we have created a script to take a backup of our important files!

### Note to Windows Users

You can set the `source` list and `target` directory to any file and directory names but you have to be a little careful in Windows. The problem is that Windows uses the backslash (`\`) as the directory separator character but Python uses backslashes to represent escape sequences!

So, you have to represent a backslash itself using an escape sequence or you have to use raw strings. For example, use `'C:\\Documents'` or `r'C:\Documents'` but do **not** use `'C:\Documents'` - you are using an unknown escape sequence `\D` !

Now that we have a working backup script, we can use it whenever we want to take a backup of the files. Linux/Unix users are advised to use the executable method as discussed earlier so that they can run the backup script anytime anywhere. This is called the **operation** phase or the **deployment** phase of the software.

The above program works properly, but (usually) first programs do not work exactly as you expect. For example, there might be problems if you have not designed the program properly or if you have made a mistake in typing the code, etc. Appropriately, you will have to go back to the design phase or you will have to debug your program.

## Second Version

The first version of our script works. However, we can make some refinements to it so that it can work better on a daily basis. This is called the **maintenance** phase of the software.

One of the refinements I felt was useful is a better file-naming mechanism - using the *time* as the name of the file within a directory with the current *date* as a directory within the main backup directory. One advantage is that your backups are stored in a hierarchical manner and therefore it is much easier to manage. Another advantage is that the length of the filenames are much shorter this way. Yet another advantage is that separate directories will help you to easily check if you have taken a backup for each day since the directory would be created only if you have taken a backup for that day.

### Example 10.2. Backup Script - The Second Version

```
#!/usr/bin/python
# Filename: backup_ver2.py

import os
import time

# 1. The files and directories to be backed up are specified in a list.
source = ['/home/swaroop/byte', '/home/swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or somethin
# 2. The backup must be stored in a main backup directory
target_dir = '/mnt/e/backup/' # Remember to change this to what you will be using

# 3. The files are backed up into a zip file.
# 4. The current day is the name of the subdirectory in the main directory
today = target_dir + time.strftime('%Y%m%d')
# The current time is the name of the zip archive
now = time.strftime('%H%M%S')

# Create the subdirectory if it isn't already there
if not os.path.exists(today):
        os.mkdir(today) # make directory
        print 'Successfully created directory', today

# The name of the zip file
target = today + os.sep + now + '.zip'

# 5. We use the zip command (in Unix/Linux) to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))

# Run the backup
if os.system(zip_command) == 0:
        print 'Successful backup to', target
else:
        print 'Backup FAILED'
```

## Output

```
$ python backup_ver2.py
Successfully created directory /mnt/e/backup/20041208
Successful backup to /mnt/e/backup/20041208/080020.zip

$ python backup_ver2.py
Successful backup to /mnt/e/backup/20041208/080428.zip
```

## How It Works

Most of the program remains the same. The changes is that we check if there is a directory with the current day as name inside the main backup directory using the `os.exists` function. If it doesn't exist, we create it using the `os.mkdir` function.

Notice the use of `os.sep` variable - this gives the directory separator according to your operating system i.e. it will be `'/'` in Linux, Unix, it will be `'\\'` in Windows and `':'` in Mac OS. Using `os.sep` instead of these characters directly will make our program portable and work across these systems.

## Third Version

The second version works fine when I do many backups, but when there are lots of backups, I am finding it hard to differentiate what the backups were for! For example, I might have made some major changes to a program or presentation, then I want to associate what those changes are with the name of the zip archive. This can be easily achieved by attaching a user-supplied comment to the name of the zip archive.

### Example 10.3. Backup Script - The Third Version (does not work!)

```
#!/usr/bin/python
# Filename: backup_ver2.py

import os
import time

# 1. The files and directories to be backed up are specified in a list.
source = ['/home/swaroop/byte', '/home/swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or somethin

# 2. The backup must be stored in a main backup directory
target_dir = '/mnt/e/backup/' # Remember to change this to what you will be using

# 3. The files are backed up into a zip file.
# 4. The current day is the name of the subdirectory in the main directory
today = target_dir + time.strftime('%Y%m%d')
# The current time is the name of the zip archive
now = time.strftime('%H%M%S')

# Take a comment from the user to create the name of the zip file
comment = raw_input('Enter a comment --> ')
if len(comment) == 0: # check if a comment was entered
        target = today + os.sep + now + '.zip'
else:
        target = today + os.sep + now + '_' +
                comment.replace(' ', '_') + '.zip'

# Create the subdirectory if it isn't already there
if not os.path.exists(today):
        os.mkdir(today) # make directory
        print 'Successfully created directory', today

# 5. We use the zip command (in Unix/Linux) to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))

# Run the backup
if os.system(zip_command) == 0:
        print 'Successful backup to', target
else:
        print 'Backup FAILED'
```

## Output

```
$ python backup_ver3.py
File "backup_ver3.py", line 25
target = today + os.sep + now + '_' +
                                    ^

SyntaxError: invalid syntax
```

## How This (does not) Work

**This program does not work!**. Python says there is a syntax error which means that the script does not satisfy the structure that Python expects to see. When we observe the error given by Python, it also tells us the place where it detected the error as well. So we start *debugging* our program from that line.

On careful observation, we see that the single logical line has been split into two physical lines but we have not specified that these two physical lines belong together. Basically, Python has found the addition operator (+) without any operand in that logical line and hence it doesn't know how to continue. Remember that we can specify that the logical line continues in the next physical line by the use of a backslash at the end of the physical line. So, we make this correction to our program. This is called **bug fixing**.

## Fourth Version

### Example 10.4. Backup Script - The Fourth Version

```
#!/usr/bin/python
# Filename: backup_ver2.py

import os, time

# 1. The files and directories to be backed up are specified in a list.
source = ['/home/swaroop/byte', '/home/swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or somethin

# 2. The backup must be stored in a main backup directory
target_dir = '/mnt/e/backup/' # Remember to change this to what you will be using

# 3. The files are backed up into a zip file.
# 4. The current day is the name of the subdirectory in the main directory
today = target_dir + time.strftime('%Y%m%d')
# The current time is the name of the zip archive
now = time.strftime('%H%M%S')

# Take a comment from the user to create the name of the zip file
comment = raw_input('Enter a comment --> ')
if len(comment) == 0: # check if a comment was entered
        target = today + os.sep + now + '.zip'
else:
        target = today + os.sep + now + '_' + \
                comment.replace(' ', '_') + '.zip'
         # Notice the backslash!

# Create the subdirectory if it isn't already there
if not os.path.exists(today):
        os.mkdir(today) # make directory
        print 'Successfully created directory', today

# 5. We use the zip command (in Unix/Linux) to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))

# Run the backup
```

```
if os.system(zip_command) == 0:
        print 'Successful backup to', target
else:
        print 'Backup FAILED'
```

## Output

```
$ python backup_ver4.py
Enter a comment --> added new examples
Successful backup to /mnt/e/backup/20041208/082156_added_new_examples.zip

$ python backup_ver4.py
Enter a comment -->
Successful backup to /mnt/e/backup/20041208/082316.zip
```

## How It Works

This program now works! Let us go through the actual enhancements that we had made in version 3. We take in the user's comments using the `raw_input` function and then check if the user actually entered something by finding out the length of the input using the `len` function. If the user has just pressed **enter** for some reason (maybe it was just a routine backup or no special changes were made), then we proceed as we have done before.

However, if a comment was supplied, then this is attached to the name of the zip archive just before the `.zip` extension. Notice that we are replacing spaces in the comment with underscores - this is because managing such filenames are much easier.

## More Refinements

The fourth version is a satisfactorily working script for most users, but there is always room for improvement. For example, you can include a *verbosity* level for the program where you can specify a `-v` option to make your program become more talkative.

Another possible enhancement would be to allow extra files and directories to be passed to the script at the command line. We will get these from the `sys.argv` list and we can add them to our `source` list using the `extend` method provided by the `list` class.

One refinement I prefer is the use of the **tar** command instead of the **zip** command. One advantage is that when you use the **tar** command along with **gzip**, the backup is much faster and the backup created is also much smaller. If I need to use this archive in Windows, then WinZip handles such `.tar.gz` files easily as well. The **tar** command is available by default on most Linux/Unix systems. Windows users can download [http://gnuwin32.sourceforge.net/packages/tar.htm] and install it as well.

The command string will now be:

```
tar = 'tar -cvzf %s %s -X /home/swaroop/excludes.txt' % (target, ' '.join(srcdir))
```

The options are explained below.

- `-c` indicates **c**reation of an archive.
- `-v` indicates **v**erbose i.e. the command should be more talkative.
- `-z` indicates the g**z**ip filter should be used.
- `-f` indicates **f**orce in creation of archive i.e. it should replace if there is a file by the same name already.
- `-X` indicates a file which contains a list of filenames which must be e**x**cluded from the backup. For example, you can specify `*~` in this file to not include any filenames ending with ~ in the backup.

### Important

The most preferred way of creating such kind of archives would be using the `zipfile` or `tarfile` module respectively. They are part of the Python Standard Library and available for you to use already. Using these libraries also avoids the use of the `os.system` which is generally not advisable to use because it is very easy to make costly mistakes using it.

However, I have been using the `os.system` way of creating a backup purely for pedagogical purposes, so that the example is simple enough to be understood by everybody but real enough to be useful.

## The Software Development Process

We have now gone through the various **phases** in the process of writing a software. These phases can be summarised as follows:

1. What (Analysis)
2. How (Design)
3. Do It (Implementation)
4. Test (Testing and Debugging)
5. Use (Operation or Deployment)
6. Maintain (Refinement)

### Important

A recommended way of writing programs is the procedure we have followed in creating the backup script - Do the analysis and design. Start implementing with a simple version. Test and debug it. Use it to ensure that it works as expected. Now, add any features that you want and continue to repeat the Do It-Test-Use cycle as many times as required. Remember, '**Software is grown, not built**'.

## Summary

We have seen how to create our own Python programs/scripts and the various stages involved in writing such programs. You may find it useful to create your own program just like we did in this chapter so that you become comfortable with Python as well as problem-solving.

Next, we will discuss object-oriented programming.

# Chapter 11. Object-Oriented Programming

## Introduction

In all our programs till now, we have designed our program around functions or blocks of statements which manipulate data. This is called the *procedure-oriented* way of programming. There is another way of organizing your program which is to combine data and functionality and wrap it inside what is called an object. This is called the *object oriented* programming paradigm. Most of the time you can use procedural programming but sometimes when you want to write large programs or have a solution that is better suited to it, you can use object oriented programming techniques.

Classes and objects are the two main aspecs of object oriented programming. A **class** creates a new *type* where **objects** are *instances* of the class. An analogy is that you can have variables of type `int` which translates to saying that variables that store integers are variables which are instances (objects) of the `int` class.

### Note for C/C++/Java/C# Programmers

Note that even integers are treated as objects (of the `int` class). This is unlike C++ and Java (before version 1.5) where integers are primitive native types. See `help(int)` for more details on the class.

C# and Java 1.5 programmers will be familiar with this concept since it is similar to the *boxing and unboxing* concept.

Objects can store data using ordinary variables that *belong* to the object. Variables that belong to an object or class are called as **fields**. Objects can also have functionality by using functions that *belong* to a class. Such functions are called **methods** of the class. This terminology is important because it helps us to differentiate between functions and variables which are separate by itself and those which belong to a class or object. Collectively, the fields and methods can be referred to as the **attributes** of that class.

Fields are of two types - they can belong to each instance/object of the class or they can belong to the class itself. They are called **instance variables** and **class variables** respectively.

A class is created using the `class` keyword. The fields and methods of the class are listed in an indented block.

## The self

Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you do **do not** give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object itself, and by convention, it is given the name `self`.

Although, you can give any name for this parameter, it is *strongly recommended* that you use the name `self` - any other name is definitely frowned upon. There are many advantages to using a standard name - any reader of your program will immediately recognize it and even specialized IDEs (Integrated Development Environments) can help you if you use `self`.

### Note for C++/Java/C# Programmers

The `self` in Python is equivalent to the `self` pointer in C++ and the `this` reference in Java and C#.

You must be wondering how Python gives the value for `self` and why you don't need to give a value for it. An example will make this clear. Say you have a class called `MyClass` and an instance of this class called `MyObject`. When you call a method of this object as `MyObject.method(arg1, arg2)`, this is automatically converted by Python into `MyClass.method(MyObject, arg1, arg2` - this is what the special `self` is all about.

This also means that if you have a method which takes no arguments, then you still have to define the method to have a `self` argument.

## Classes

The simplest class possible is shown in the following example.

## Creating a Class

**Example 11.1. Creating a Class**

```
#!/usr/bin/python
# Filename: simplestclass.py

class Person:
        pass # An empty block

p = Person()
print p
```

### Output

```
$ python simplestclass.py
<__main__.Person instance at 0xf6fcb18c>
```

### How It Works

We create a new class using the `class` statement followed by the name of the class. This follows an indented block of statements which form the body of the class. In this case, we have an empty block which is indicated using the `pass` statement.

Next, we create an object/instance of this class using the name of the class followed by a pair of parentheses. (We will learn more about instantiation in the next section). For our verification, we confirm the type of the variable by simply printing it. It tells us that we have an instance of the `Person` class in the `__main__` module.

Notice that the address of the computer memory where your object is stored is also printed. The address will have a different value on your computer since Python can store the object wherever it finds space.

## object Methods

We have already discussed that classes/objects can have methods just like functions except that we have an extra `self` variable. We will now see an example.

## Using Object Methds

**Example 11.2. Using Object Methods**

```
#!/usr/bin/python
# Filename: method.py

class Person:
        def sayHi(self):
                print 'Hello, how are you?'

p = Person()
p.sayHi()

# This short example can also be written as Person().sayHi()
```

### Output

```
$ python method.py
Hello, how are you?
```

### How It Works

Here we see the `self` in action. Notice that the `sayHi` method takes no parameters but still has the `self` in the function definition.

## The __init__ method

There are many method names which have special significance in Python classes. We will see the significance of the `__init__` method now.

The `__init__` method is run as soon as an object of a class is instantiated. The method is useful to do any *initialization* you want to do with your object. Notice the double underscore both in the beginning and at the end in the name.

## Using the __init__ method

**Example 11.3. Using the __init__ method**

```
#!/usr/bin/python
# Filename: class_init.py

class Person:
        def __init__(self, name):
                self.name = name
        def sayHi(self):
                print 'Hello, my name is', self.name

p = Person('Swaroop')
p.sayHi()

# This short example can also be written as Person('Swaroop').sayHi()
```

### Output

```
$ python class_init.py
Hello, my name is Swaroop
```

### How It Works

Here, we define the `__init__` method as taking a parameter `name` (along with the usual `self`). Here, we just create a new field also called `name`. Notice these are two different variables even though they have the same name. The dotted notation allows us to differentiate between them.

Most importantly, notice that we do not explicitly call the `__init__` method but pass the arguments in the parentheses following the class name when creating a new instance of the class. This is the special significance of this method.

Now, we are able to use the `self.name` field in our methods which is demonstrated in the `sayHi` method.

#### Note for C++/Java/C# Programmers

The `__init__` method is analogous to a *constructor* in C++, C# or Java.

## Class and Object Variables

We have already discussed the functionality part of classes and objects, now we'll see the data part of it. Actually, they are nothing but ordinary variables which are *bound* to the classes and objects **namespaces** i.e. the names are valid within the context of these classes and objects only.

There are two types of *fields* - class variables and object variables which are classified depending on whether the class or the object *owns* the variables respectively.

*Class variables* are shared in the sense that they are accessed by all objects (instances) of that class. There is only copy of the class variable and when any one object makes a change to a class variable, the change is reflected in all the other instances as well.

*Object variables* are owned by each individual object/instance of the class. In this case, each object has its own copy of the field i.e. they are not shared and are not related in any way to the field by the samen name in a different instance of the same class. An example will make this easy to understand.

## Using Class and Object Variables

### Example 11.4. Using Class and Object Variables

```
#!/usr/bin/python
# Filename: objvar.py

class Person:
        '''Represents a person.'''
        population = 0

        def __init__(self, name):
                '''Initializes the person's data.'''
                self.name = name
                print '(Initializing %s)' % self.name

                # When this person is created, he/she
                # adds to the population
                Person.population += 1

        def __del__(self):
                '''I am dying.'''
                print '%s says bye.' % self.name

                Person.population -= 1

                if Person.population == 0:
                        print 'I am the last one.'
                else:
                        print 'There are still %d people left.' % Person.populatio

        def sayHi(self):
                '''Greeting by the person.

                Really, that's all it does.'''
                print 'Hi, my name is %s.' % self.name

        def howMany(self):
                '''Prints the current population.'''
                if Person.population == 1:
                        print 'I am the only person here.'
                else:
                        print 'We have %d persons here.' % Person.population

swaroop = Person('Swaroop')
swaroop.sayHi()
swaroop.howMany()

kalam = Person('Abdul Kalam')
kalam.sayHi()
kalam.howMany()

swaroop.sayHi()
swaroop.howMany()
```

## Output

```
$ python objvar.py
(Initializing Swaroop)
Hi, my name is Swaroop.
I am the only person here.
(Initializing Abdul Kalam)
Hi, my name is Abdul Kalam.
We have 2 persons here.
Hi, my name is Swaroop.
We have 2 persons here.
Abdul Kalam says bye.
There are still 1 people left.
Swaroop says bye.
I am the last one.
```

## How It Works

This is a long example but helps demonstrate the nature of class and object variables. Here, `population` belongs to the `Person` class and hence is a class variable. The `name` variable belongs to the object (it is assigned using `self`) and hence is an object variable.

Thus, we refer to the `population` class variable as `Person.population` and not as `self.population`. Note that an object variable with the same name as a class variable will hide the class variable! We refer to the object variable `name` using `self.name` notation in the methods of that object. Remember this simple difference between class and object variables.

Observe that the `__init__` method is used to initialize the `Person` instance with a name. In this method, we increase the `population` count by 1 since we have one more person being added. Also observe that the values of `self.name` is specific to each object which indicates the nature of object variables.

Remember, that you must refer to the variables and methods of the same object using the `self` variable **only**. This is called an *attribute reference*.

In this program, we also see the use of **docstrings** for classes as well as methods. We can access the class docstring at runtime using `Person.__doc__` and the method docstring as `Person.sayHi.__doc__`

Just like the `__init__` method, there is another special method `__del__` which is called when an object is going to die i.e. it is no longer being used and is being returned to the system for reusing that piece of memory. In this method, we simply decrease the `Person.population` count by 1.

The `__del__` method is run when the object is no longer in use and there is no guarantee *when* that method will be run. If you want to explicitly do this, you just have to use the `del` statement which we have used in previous examples.

### Note for C++/Java/C# Programmers

All class members (including the data members) are *public* and all the methods are *virtual* in Python.

One exception: If you use data members with names using the *double underscore prefix* such as `__privatevar`, Python uses name-mangling to effectively make it a private variable.

Thus, the convention followed is that any variable that is to be used only within the class or object should begin with an underscore and all other names are public and can be used by other classes/objects. Remember that this is only a convention and is not enforced by Python (except for the double underscore prefix).

Also, note that the `__del__` method is analogous to the concept of a *destructor*.

## Inheritance

One of the major benefits of object oriented programming is **reuse** of code and one of the ways this is achieved is through the *inheritance* mechanism. Inheritance can be best imagined as implementing a *type and subtype* relationship between classes.

Suppose you want to write a program which has to keep track of the teachers and students in a college. They have some common characteristics such as name, age and address. They also have specific characteristics such as salary, courses and leaves for teachers and, marks and fees for students.

You can create two independent classes for each type and process them but adding a new common characteristic would mean adding to both of these independent classes. This quickly becomes unwieldy.

A better way would be to create a common class called `SchoolMember` and then have the teacher and student classes *inherit* from this class i.e. they will become sub-types of this type (class) and then we can add specific characteristics to these sub-types.

There are many advantages to this approach. If we add/change any functionality in `SchoolMember`, this is automatically reflected in the subtypes as well. For example, you can add a new ID card field for both teachers and students by simply adding it to the SchoolMember class. However, changes in the subtypes do not affect other subtypes. Another advantage is that if you can refer to a teacher or student object as a `SchoolMember` object which could be useful in some situations such as counting of the number of school members. This is called **polymorphism** where a sub-type can be substituted in any situation where a parent type is expected i.e. the object can be treated as an instance of the parent class.

Also observe that we *reuse* the code of the parent class and we do not need to repeat it in the different classes as we would have had to in case we had used independent classes.

The `SchoolMember` class in this situation is known as the *base class* or the *superclass*. The `Teacher` and `Student` classes are called the *derived classes* or *subclasses*.

We will now see this example as a program.

## Using Inheritance

### Example 11.5. Using Inheritance

```
#!/usr/bin/python
# Filename: inherit.py
```

```
class SchoolMember:
	'''Represents any school member.'''
	def __init__(self, name, age):
		self.name = name
		self.age = age
		print '(Initialized SchoolMember: %s)' % self.name

	def tell(self):
		'''Tell my details.'''
		print 'Name:"%s" Age:"%s"' % (self.name, self.age),

class Teacher(SchoolMember):
	'''Represents a teacher.'''
	def __init__(self, name, age, salary):
		SchoolMember.__init__(self, name, age)
		self.salary = salary
		print '(Initialized Teacher: %s)' % self.name

	def tell(self):
		SchoolMember.tell(self)
		print 'Salary: "%d"' % self.salary

class Student(SchoolMember):
	'''Represents a student.'''
	def __init__(self, name, age, marks):
		SchoolMember.__init__(self, name, age)
		self.marks = marks
		print '(Initialized Student: %s)' % self.name

	def tell(self):
		SchoolMember.tell(self)
		print 'Marks: "%d"' % self.marks

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 22, 75)

print # prints a blank line

members = [t, s]
for member in members:
	member.tell() # works for both Teachers and Students
```

## Output

```
$ python inherit.py
(Initialized SchoolMember: Mrs. Shrividya)
(Initialized Teacher: Mrs. Shrividya)
(Initialized SchoolMember: Swaroop)
(Initialized Student: Swaroop)

Name:"Mrs. Shrividya" Age:"40" Salary: "30000"
Name:"Swaroop" Age:"22" Marks: "75"
```

## How It Works

To use inheritance, we specify the base class names in a tuple following the class name in the class definition. Next, we observe that the `__init__` method of the base class is explicitly called using the `self` variable so that we can initialize the base class part of the object. This is very important to remember - Python does not automatically call the constructor of the base class, you have to explicitly call it yourself.

We also observe that we can call methods of the base class by prefixing the class name to the method call and then pass in the `self` variable along with any arguments.

Notice that we can treat instances of `Teacher` or `Student` as just instances of the `SchoolMember` when we use the `tell` method of the `SchoolMember` class.

Also, observe that the `tell` method of the subtype is called and not the `tell` method of the `SchoolMember` class. One way to understand this is that Python *always* starts looking for methods in the type, which in this case it does. If it could not find the method, it starts looking at the methods belonging to its base classes one by one in the order they are specified in the tuple in the class definition.

A note on terminology - if more than one class is listed in the inheritance tuple, then it is called *multiple inheritance*.

## Summary

We have now explored the various aspects of classes and objects as well as the various terminologies associated with it. We have also seen the benefits and pitfalls of object-oriented programming. Python is highly object-oriented and understanding these concepts carefully will help you a lot in the long run.

Next, we will learn how to deal with input/output and how to access files in Python.

# Chapter 12. Input/Output

There will be lots of times when you want your program to interact with the user (which could be yourself). You would want to take input from the user and then print some results back. We can achieve this using the `raw_input` and `print` statements respectively. For output, we can also use the various methods of the `str` (string) class. For example, you can use the `rjust` method to get a string which is right justified to a specified width. See `help(str)` for more details.

Another common type of input/output is dealing with files. The ability to create, read and write files is essential to many programs and we will explore this aspect in this chapter.

## Files

You can open and use files for reading or writing by creating an object of the `file` class and using its `read`, `readline` or `write` methods appropriately to read from or write to the file. The ability to read or write to the file depends on the mode you have specified for the file opening. Then finally, when you are finished with the file, you call the `close` method to tell Python that we are done using the file.

## Using file

**Example 12.1. Using files**

```
#!/usr/bin/python
# Filename: using_file.py

poem = '''\
Programming is fun
When the work is done
if you wanna make your work also fun:
        use Python!
'''

f = file('poem.txt', 'w') # open for 'w'riting
f.write(poem) # write text to file
f.close() # close the file

f = file('poem.txt') # if no mode is specified, 'r'ead mode is assumed by default
while True:
        line = f.readline()
        if len(line) == 0: # Zero length indicates EOF
                break
        print line, # Notice comma to avoid automatic newline added by Python
f.close() # close the file
```

### Output

```
$ python using_file.py
```

```
Programming is fun
When the work is done
if you wanna make your work also fun:
        use Python!
```

### How It Works

First, we create an instance of the `file` class by specifying the name of the file and the mode in which we want to open the file. The mode can be a read mode (`'r'`), write mode (`'w'`) or append mode (`'a'`). There are actually many more modes available and `help(file)` will give you more details about them.

We first open the file in write mode and use the `write` method of the `file` class to write to the file and then we finally `close` the file.

Next, we open the same file again for reading. If we don't specify a mode, then the read mode is the default one. We read in each line of the file using the `readline` method, in a loop. This method returns a complete line including the newline character at the end of the line. So, when an *empty* string is returned, it indicates that the end of the file has been reached and we stop the loop.

Notice that we use a comma with the `print` statement to suppress the automatic newline that the `print` statement adds because the line that is read from the file already ends with a newline character. Then, we finally `close` the file.

Now, see the contents of the `poem.txt` file to confirm that the program has indeed worked properly.

## Pickle

Python provides a standard module called `pickle` using which you can store **any** Python object in a file and then get it back later intact. This is called storing the object *persistently*.

There is another module called `cPickle` which functions exactly same as the `pickle` module except that it is written in the C language and is (upto 1000 times) faster. You can use either of these modules, although we will be using the `cPickle` module here. Remember though, that we refer to both these modules as simply the `pickle` module.

## Pickling and Unpickling

**Example 12.2. Pickling and Unpickling**

```
#!/usr/bin/python
# Filename: pickling.py

import cPickle as p
#import pickle as p

shoplistfile = 'shoplist.data' # the name of the file where we will store the obje

shoplist = ['apple', 'mango', 'carrot']

# Write to the file
```

```
f = file(shoplistfile, 'w')
p.dump(shoplist, f) # dump the object to a file
f.close()

del shoplist # remove the shoplist

# Read back from the storage
f = file(shoplistfile)
storedlist = p.load(f)
print storedlist
```

### Output

```
$ python pickling.py
['apple', 'mango', 'carrot']
```

### How It Works

First, notice that we use the `import..as` syntax. This is handy since we can use a shorter name for a module. In this case, it even allows us to switch to a different module (`cPickle` or `pickle`) by simply changing one line! In the rest of the program, we simply refer to this module as `p`.

To store an object in a file, first we open a `file` object in write mode and store the object into the open file by calling the `dump` function of the pickle module. This process is called *pickling*.

Next, we retrieve the object using the `load` function of the `pickle` module which returns the object. This process is called *unpickling*.

## Summary

We have discussed various types of input/output and also file handling and using the pickle module.

Next, we will explore the concept of exceptions.

# Chapter 13. Exceptions

Exceptions occur when certain *exceptional* situations occur in your program. For example, what if you are going to read a file and the file does not exist? Or what if you accidentally deleted it when the program was running? Such situations are handled using **exceptions**.

What if your program had some invalid statements? This is handled by Python which **raises** its hands and tells you there is an **error**.

## Errors

Consider a simple `print` statement. What if we misspelt `print` as `Print`? Note the capitalization. In this case, Python *raises* a syntax error.

```
>>> Print 'Hello World'
    File "<stdin>", line 1
      Print 'Hello World'
                        ^
SyntaxError: invalid syntax

>>> print 'Hello World'
Hello World
```

Observe that a `SyntaxError` is raised and also the location where the error was detected is printed. This is what an *error handler* for this error does.

## Try..Except

We will **try** to read input from the user. Press **Ctrl-d** and see what happens.

```
>>> s = raw_input('Enter something --> ')
Enter something --> Traceback (most recent call last):
  File "<stdin>", line 1, in ?
EOFError
```

Python raises an error called `EOFError` which basically means it found an *end of file* when it did not expect to (which is represented by **Ctrl-d**)

Next, we will see how to handle such errors.

### Handling Exceptions

We can handle exceptions using the `try..except` statement. We basically put our usual statements within the try-block and put all our error handlers in the except-block.

**Example 13.1. Handling Exceptions**

```
#!/usr/bin/python
# Filename: try_except.py

import sys

try:
        s = raw_input('Enter something --> ')
except EOFError:
        print '\nWhy did you do an EOF on me?'
        sys.exit() # exit the program
except:
        print '\nSome error/exception occurred.'
        # here, we are not exiting the program

print 'Done'
```

## Output

```
$ python try_except.py
Enter something -->
Why did you do an EOF on me?

$ python try_except.py
Enter something --> Python is exceptional!
Done
```

## How It Works

We put all the statements that might raise an error in the `try` block and then handle all the errors and exceptions in the `except` clause/block. The `except` clause can handle a single specified error or exception, or a parenthesized list of errors/exceptions. If no names of errors or exceptions are supplied, it will handle *all* errors and exceptions. There has to be at least one `except` clause associated with every `try` clause.

If any error or exception is not handled, then the default Python handler is called which just stops the execution of the program and prints a message. We have already seen this in action.

You can also have an `else` clause associated with a `try..catch` block. The `else` clause is executed if no exception occurs.

We can also get the exception object so that we can retrieve additional information about the exception which has occurred. This is demonstrated in the next example.

# Raising Exceptions

You can *raise* exceptions using the `raise` statement. You also have to specify the name of the error/exception and the exception object that is to be *thrown* along with the exception. The error or exception that you can arise should be class which directly or indirectly is a derived class of the `Error` or `Exception` class respectively.

## How To Raise Exceptions

### Example 13.2. How to Raise Exceptions

```
#!/usr/bin/python
# Filename: raising.py

class ShortInputException(Exception):
	'''A user-defined exception class.'''
	def __init__(self, length, atleast):
		Exception.__init__(self)
		self.length = length
		self.atleast = atleast

try:
	s = raw_input('Enter something --> ')
	if len(s) < 3:
		raise ShortInputException(len(s), 3)
	# Other work can continue as usual here
except EOFError:
	print '\nWhy did you do an EOF on me?'
except ShortInputException, x:
	print 'ShortInputException: The input was of length %d, \
		was expecting at least %d' % (x.length, x.atleast)
else:
	print 'No exception was raised.'
```

## Output

```
$ python raising.py
Enter something -->
Why did you do an EOF on me?

$ python raising.py
Enter something --> ab
ShortInputException: The input was of length 2, was expecting at least 3

$ python raising.py
Enter something --> abc
No exception was raised.
```

## How It Works

Here, we are creating our own exception type although we could've used any predefined exception/error for demonstration purposes. This new exception type is the `ShortInputException` class. It has two fields - `length` which is the length of the given input, and `atleast` which is the minimum length that the program was expecting.

In the `except` clause, we mention the class of error as well as the variable to hold the corresponding error/exception object. This is analogous to parameters and arguments in a function call. Within this particular `except` clause, we use the `length` and `atleast` fields of the exception object to print an appropriate message to the user.

## Try..Finally

What if you were reading a file and you wanted to close the file whether or not an exception was raised? This can be done using the `finally` block. Note that you can use an `except` clause along with a `finally` block for the same corresponding `try` block. You will have to embed one within another if you want to use both.

## Using Finally

**Example 13.3. Using Finally**

```
#!/usr/bin/python
# Filename: finally.py

import time

try:
        f = file('poem.txt')
        while True: # our usual file-reading idiom
                line = f.readline()
                if len(line) == 0:
                        break
                time.sleep(2)
                print line,
finally:
        f.close()
        print 'Cleaning up...closed the file'
```

### Output

```
$ python finally.py
Programming is fun
When the work is done
Cleaning up...closed the file
Traceback (most recent call last):
  File "finally.py", line 12, in ?
    time.sleep(2)
KeyboardInterrupt
```

## How It Works

We do the usual file-reading stuff, but I've arbitrarily introduced a way of sleeping for 2 seconds before printing each line using the `time.sleep` method. The only reason is so that the program runs slowly (Python is very fast by nature). When the program is still running, press **Ctrl-c** to interrupt/cancel the program.

Observe that a `KeyboardInterrupt` exception is thrown and the program exits, but before the program exits, the finally clause is executed and the file is closed.

# Summary

We have discussed the usage of the `try..except` and `try..finally` statements. We have seen how to create our own exception types and how to raise exceptions as well.

Next, we will explore the Python Standard Library.

# Chapter 14. The Python Standard Library

## Introduction

The Python Standard Library is available with every Python installation. It contains a huge number of very useful modules. It is important that you become familiar with the Python Standard Library since most of your problems can be solved more easily and quickly if you are familiar with this library of modules.

We will explore some of the commonly used modules in this library. You can find complete details for all of the modules in the Python Standard Library in the 'Library Reference' section in the documentation that comes with your Python installation.

## The sys module

The `sys` module contains system-specific functionality. we have already seen that the `sys.argv` list contains the command-line arguments.

### Command Line Arguments

**Example 14.1. Using sys.argv**

```
#!/usr/bin/python
# Filename: cat.py

import sys

def readfile(filename):
        '''Print a file to the standard output.'''
        f = file(filename)
        while True:
                line = f.readline()
                if len(line) == 0:
                        break
                print line, # notice comma
        f.close()

# Script starts from here
if len(sys.argv) < 2:
        print 'No action specified.'
        sys.exit()

if sys.argv[1].startswith('--'):
        option = sys.argv[1][2:]
        # fetch sys.argv[1] but without the first two characters
        if option == 'version':
                print 'Version 1.2'
        elif option == 'help':
                print '''\
This program prints files to the standard output.
```

```
Any number of files can be specified.
Options include:
  --version : Prints the version number
  --help    : Display this help'''
        else:
                print 'Unknown option.'
        sys.exit()
else:
        for filename in sys.argv[1:]:
                readfile(filename)
```

## Output

```
$ python cat.py
No action specified.

$ python cat.py --help
This program prints files to the standard output.
Any number of files can be specified.
Options include:
  --version : Prints the version number
  --help    : Display this help

$ python cat.py --version
Version 1.2

$ python cat.py --nonsense
Unknown option.

$ python cat.py poem.txt
Programming is fun
When the work is done
if you wanna make your work also fun:
        use Python!
```

## How It Works

This program tries to mimic the **cat** command familiar to Linux/Unix users. You just speicfy the names of some text files and it will print them to the output.

When a Python program is run i.e. not an interactive mode, there is always at least one item in the `sys.argv` list which is the name of the current program being run and is available as `sys.argv[0]` since Python starts counting from 0. Other command line arguments follow this item.

To make the program user-friendly we have supplied certain options that the user can specify to learn more about the program. We use the first argument to check if any options have been specified to our program. If the `--version` option is used, the version number of the program is printed. Similarly, when the `--help` option is specified, we give a bit of explanation about the program. We make use of the `sys.exit` function to exit the running program. As always, see `help(sys.exit)` for more details.

When no options are specified and filenames are passed to the program, it simply prints out each line of each file, one after the other in the order specified on the command line.

As an aside, the name **cat** is short for *concatenate* which is basically what this program does - it can print out a file or attach/concatenate two or more files together in the output.

## More sys

The `sys.version` string gives you information about the version of Python that you have installed. The `sys.version_info` tuple gives an easier way of enabling Python-version specific parts of your program.

```
[swaroop@localhost code]$ python
>>> import sys
>>> sys.version
'2.3.4 (#1, Oct 26 2004, 16:42:40) \n[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)]'
>>> sys.version_info
(2, 3, 4, 'final', 0)
```

For experienced programmers, other items of interest in the `sys` module include `sys.stdin`, `sys.stdout` and `sys.stderr` which correspond to the standard input, standard output and standard error streams of your program respectively.

## The os module

This module represents generic **o**perating **s**ystem functionality. This module is especially important if you want to make your programs platform-independent i.e. it allows the program to be written such that it will run on Linux as well as Windows without any problems and without requiring changes. An example of this is using the `os.sep` variable instead of the operation system-specific path separator.

Some of the more useful parts of the `os` module are listed below Most of them are self-explanatory.

- The `os.name` string specifies which platform you are using, such as `'nt'` for Windows and `'posix'` for Linux/Unix users.
- The `os.getcwd()` function gets the current working directory i.e. the path of the directory from which the curent Python script is working.
- The `os.getenv()` and `os.putenv()` functions are used to get and set environment variables respectively.
- The `os.listdir()` function returns the name of all files and directories in the specified directory.
- The `os.remove()` function is used to delete a file.
- The `os.system()` function is used to run a shell command.
- The `os.linesep` string gives the line terminator used in the current platform. For example, Windows uses `'\r\n'`, Linux uses `'\n'` and Mac uses `'\r'`.
- The `os.path.split()` function returns the directory name and file name of the path.

```
>>> os.path.split('/home/swaroop/byte/code/poem.txt')
('/home/swaroop/byte/code', 'poem.txt')
```

- The `os.path.isfile()` and the `os.path.isdir()` functions check if the given path refers to a file or directory respectively. Similarly, the `os.path.exists()` function is used to check if a given path actually exists.

You can explore the Python Standard Documentation for more details on these functions and variables. You can use `help(sys)`, etc. as well.

## Summary

We have seen some of the functionality of the `sys` module and `sys` modules in the Python Standard Library. You should explore the Python Standard Documentation to find out more about these and other modules as well.

Next, we will cover various aspects of Python that will make our tour of Python more *complete*.

# Chapter 15. More Python

Till now, we have covered majority of the various aspects of Python that you will use. In this chapter, we will cover some more aspects that will make our knowledge of Python more *complete*.

## Special Methods

There are certain special methods which have special significance in classes such as the `__init__` and `__del__` methods whose significance we have already seen.

Generally, special methods are used to mimic certain behavior. For example, if you want to use the `x[key]` indexing operation for your class (just like you use for lists and tuples) then just implement the `__getitem__()` method and your job is done. If you think about it, this is what Python does for the `list` class itself!

Some useful special methods are listed in the following table. If you want to know about all the special methods, then a huge list is available in the Python Reference Manual.

**Table 15.1. Some Special Methods**

| Name | Explanation |
|---|---|
| __init__(self, ...) | This method is called just before the newly created object is returned for usage. |
| __del__(self) | Called just before the object is destroyed |
| __str__(self) | Called when we use the `print` statement with the object or when `str()` is used. |
| __lt__(self, other) | Called when the *less than* operator ( < ) is used. Similarly, there are special methods for all the operators (+, >, etc.) |
| __getitem__(self, key) | Called when `x[key]` indexing operation is used. |
| __len__(self) | Called when the built-in `len()` function is used for the sequence object. |

## Single Statement Blocks

By now, you should have firmly understood that each block of statements is set apart from the rest by its own indentation level. Well, this is true for the most part but it is not 100% accurate. If your block of statements contains only one single statement, then you can specify it on the same line of, say, a conditional statement or looping statement. The following example should make this clear:

```
>>> flag = True
>>> if flag: print 'Yes'
...
Yes
```

As we can see, the single statement is used in-place and not as a separate block. Although, you can use this for making your program *smaller*, I **strongly** recommend that you do not use this short-cut method except for error checking, etc. One major reason is that it will be much easier to add an extra statement if you are using proper indentation.

Also notice that when the Python interpreter is used in interactive mode, it helps you enter the statements by changing prompts appropriately. In the aboe case, after you entered the keyword `if`, it changes the prompt to `...` to indicate that the statement is not yet complete. When we do complete the statement in this manner, we press **enter** to confirm that the statement is complete. Then, Python finishes executing the whole statement and returns to the old prompt waiting for the next input.

## List Comprehension

List comprehensions are used to derive a new list from an existing list. For example, you have a list of numbers and you want to get a corresponding list with all the numbers multiplied by 2 but only when the number itself is greater than 2. List comprehensions are ideal for such situations.

## Using List Comprehensions

**Example 15.1. Using List Comprehensions**

```
#!/usr/bin/python
# Filename: list_comprehension.py

listone = [2, 3, 4]
listtwo = [2*i for i in listone if i > 2]
print listtwo
```

### Output

```
$ python list_comprehension.py
[6, 8]
```

### How It Works

Here, we derive a new list by specifying the manipulation to be done (`2*i`) when some condition is satisfied (`if i > 2`). Note that the original list remains unmodified. Many a time, we use loops to process each element of a list, the same can be achieved using list comprehensions in a more precise, compact and explicit manner.

## Receiving Tuples and Lists in Functions

There is a special way of receiving parameters to a function as a tuple or a dictionary using the `*` or `**` prefix respectively. This is useful when taking variable number of arguments in the function.

```
>>> def powersum(power, *args):
...     '''Return the sum of each argument raised to specified power.'''
...     total = 0
...     for i in args:
...             total += pow(i, power)
...     return total
...
>>> powersum(2, 3, 4)
25

>>> powersum(2, 10)
100
```

Due to the `*` prefix on the `args` variable, all extra arguments passed to the function are stored in `args` as a tuple. If a `**` prefix had been used instead, the extra parameters would be considered to be key/value pairs of a dictionary.

## Lambda Forms

A `lambda` statement is used to create new function objects and then return them at runtime.

## Using Lambda Forms

**Example 15.2. Using Lambda Forms**

```
#!/usr/bin/python
# Filename: lambda.py

def make_repeater(n):
        return lambda s: s * n

twice = make_repeater(2)

print twice('word')
print twice(5)
```

### Output

```
$ python lambda.py
wordword
10
```

### How It Works

Here, we use a function `make_repeater` to create new function objects at runtime and return it. A `lambda` statement is used to create the function object. Essentially, the `lambda` takes a parameter followed by a single expression only which becomes the body of the function and the value of this expression is returned by the new function. Note that even a `print` statement cannot be used inside a lambda form, only expressions.

## The exec and eval statements

The `exec` statement is used to execute Python statements which are stored in a string or file. For example, we can generate a string containing Python code at runtime and then execute these statements using the `exec` statement. A simple example is shown below.

```
>>> exec 'print "Hello World"'
Hello World
```

The `eval` statement is used to evaluate valid Python expressions which are stored in a string. A simple example is shown below.

```
>>> eval('2*3')
6
```

## The assert statement

The `assert` statement is used to assert that something is true. For example, if you are very sure that you will have at least one element in a list you are using and want to check this, and raise an error if it is not true, then `assert` statement is ideal in this situation. When the assert statement fails, an `AssertionError` is raised.

```
>>> mylist = ['item']
>>> assert len(mylist) >= 1
>>> mylist.pop()
'item'
>>> assert len(mylist) >= 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AssertionError
```

## The repr function

The `reprt` function is used to obtain a canonical string representation of the object. Backticks (also called conversion or reverse quotes) do the same thing. Note that you will have `eval(repr(object)) == object` most of the time.

```
>>> i = []
>>> i.append('item')
>>> `i`
"['item']"
>>> repr(i)
"['item']"
```

Basically, the `repr` function or the backticks are used to obtain a printable representation of the object. you can control what your objects return for the `repr` function by defining the `__repr__` method in your class.

## Summary

We have covered some more features of Python in this chapter and yet you can be sure we haven't covered all the features of Python. However, at this stage, we have covered most of what you are ever going to use in practice. This is sufficient for you to get started with whatever programs you are going to create.

Next, we will discuss how to explore Python further.

# Chapter 16. What Next?

If you have read this book thoroughly till now and practiced writing a lot of programs, then you must have become comfortable and familiar with Python. You have probably created some Python programs to try out stuff and to exercise your Python skills as well. If you have not done it already, you should. The question now is 'What Next?'.

I would suggest that you tackle this problem: create your own command-line *address-book* program using which you can add, modify, delete or search for your contacts such as friends, family and colleagues and their information such as email address and/or phone number. Details must be stored for later retrieval.

This is fairly easy if you think about it in terms of all the various stuff that we have come across till now. If you still want directions on how to proceed, then here's a hint.

**Hint. (You shouldn't be reading this).** Create a class to represent the person's information. Use a dictionary to store person objects with their name as the key. Use the cPickle module to store the objects persistently on your hard disk. Use the dictionary built-in methods to add, delete and modify the persons.

Once you are able to do this, you can claim to be a Python programmer. Now, immediately send me a mail thanking me for this great book ;-) . This step is optional but recommended.

Here are some ways to continue your journey with Python:

## Graphical Software

**GUI** Libraries using Python - you need these to create your own graphical programs using Python. You can create your own IrfanView or Kuickshow or anything like that using the GUI libraries with their Python bindings. Bindings are what allow you to write programs in Python and use the libraries which are themselves written in C or C++ or other languages.

There are lots of choices for GUI using Python:

- **PyQt.** This is the Python binding for the Qt toolkit which is the foundation upon which the KDE is built. Qt is extremely easy to use and very powerful especially due to the Qt Designer and the amazing Qt documentation. You can use it for free on Linux but you will have to pay for it if you want to use it on Windows. PyQt is free if you want to create free (GPL'ed) software on Linux/Unix and paid if you want to create proprietary software. A good resource on PyQt is 'GUI Programming with Python: Qt Edition' [http://www.opendocs.org/pyqt/]. See the official homepage [http://www.riverbankcomputing.co.uk/pyqt/index.php] for more details.
- **PyGTK.** This is the Python binding for the GTK+ toolkit which is the foundation upon which GNOME is built. GTK+ has many quirks in usage but once you become comfortable, you can create GUI apps fast. The Glade graphical interface designer is indispensable. The documentation is yet to improve. GTK+ works well on Linux but its port to Windows is incomplete. You can create both free as well as proprietary software using GTK+. See the official homepage [http://www.pygtk.org/] for more details.
- **wxPython.** This is the Python bindings for the wxWidgets toolkit. wxPython has a learning curve associated with it. However, it is very portable and runs on Linux, Windows, Mac and even embedded platforms. There are many IDEs available for wxPython which include GUI designers as well such as SPE (Stani's Python Editor) [http://spe.pycs.net/] and the wxGlade [http://wxglade.sourceforge.net/] GUI builder. You can create free as well as proprietary software using wxPython. See the official homepage [http://www.wxpython.org/] for more details.

- **TkInter.** This is one of the oldest GUI toolkits in existence. If you have used IDLE, you have seen a TkInter program at work. The documentation for TkInter at PythonWare.org [http://www.pythonware.com/library/tkinter/introduction/index.htm] is comprehensive. TkInter is portable and works on both Linux/Unix as well as Windows. Importantly, TkInter is part of the standard Python distribution.
- For more choices, see the GuiProgramming wiki page at Python.org [http://www.python.org/cgi-bin/moinmoin/GuiProgramming]

## Summary of GUI Tools

Unfortunately, there is no one standard GUI tool for Python. I suggest that you choose one of the above tools depending on your situation. The first factor is whether you are willing to pay to use any of the GUI tools. The second factor is whether you want the program to run on Linux or Windows or both. The third factor is whether you are a KDE or GNOME user on Linux.

### Future Chapters

I am contemplating writing 1 or 2 chapters for this book on GUI Programming. I will be probably be choosing wxPython as the choice of toolkit. If you would like to present your views on the subject, please join the byte-of-python mailing list [http://lists.ibiblio.org/mailman/listinfo/byte-of-python] where readers discuss with me on what improvements can be made to the book.

## Explore More

- The **Python Standard Library** is an extensive library. Most of the time, this library will have what you are looking for. This is referred to as the 'batteries included' philosophy of Python. I highly recommend that you go through the Python Standard Documentation [http://docs.python.org/] before you proceed to start writing large Python programs.
- Python.org [http://www.python.org/] - the official homepage of the Python programming language. You will find the latest versions of the Python language and interpreter here. There are also various mailing lists where active discussions on various aspects of Python take place.
- **comp.lang.python** is the usenet newsgroup where discussion about this language takes place. You can post your doubts and queries to this newsgroup. You can access this online using Google Groups [http://groups.google.com/groups?hl=en&lr=&ie=UTF-8&group=comp.lang.python] or join the mailing list [http://mail.python.org/mailman/listinfo/python-list] which is just a mirror of the newsgroup.
- Python Cookbook [http://aspn.activestate.com/ASPN/Python/Cookbook/] is an extremely valuable collection of recipes or tips on how to solve certain kinds of problems using Python. This is a must-read for every Python user.
- Charming Python [http://gnosis.cx/publish/tech_index_cp.html] is an excellent series of Python-related articles by David Mertz.
- Dive Into Python [http://www.diveintopython.org/] is a very good book for experienced Python programmers. If you have thoroughly read the current book you are reading, then I would highly recommend that you read 'Dive Into Python' next. It covers a range of topics including XML Processing, Unit Testing and Functional Programming.
- Jython [http://www.jython.org/] is an implementation of the Python interpreter in the Java language. This means that you can write programs in Python and use the Java libraries as well! Jython is a

stable and mature software. If you are a Java programmer as well, I highly recommend that you give Jython a try.

- IronPython [http://www.ironpython.com/] is an implementation of the Python interpreter in C# language and can run on the .NET / Mono / DotGNU platform. This means that you can write programs in Python and use the .NET Libraries and other libraries provided by these 3 platforms as well! IronPython is still pre-alpha software and is suitable only for experimenting as of now. Jim Hugunin, who wrote IronPython has joined Microsoft and will be working towards a full version of IronPython in future.
- Lython [http://www.caddr.com/code/lython/] is a Lisp frontend to the Python language. It is similar to Common Lisp and compiles directly to Python bytecode which means that it will interoperate with our usual Python code.
- There are many many more resources on Python. Interesting ones are Daily Python-URL! [http://www.pythonware.com/daily/] which keeps you up to date on the latest Python happenings, Vaults of Parnassus [http://www.vex.net/parnassus/], ONLamp.com Python DevCenter [http://www.onlamp.com/python/], dirtSimple.org [http://dirtsimple.org/], Python Notes [http://pythonnotes.blogspot.com/] and many many more.

## Summary

We have now come to the end of this book but, as they say, this is the *the beginning of the end*!. You are now an avid Python user and you are no doubt ready to solve many problems using Python. You can start automating your computer to do all kinds of previously unimaginable things or write your own games and much much more. So, get started!

# Appendix A. Free/Libré and Open Source Software (FLOSS)

FLOSS is based on the concept of a community, which itself is based on the concept of sharing, and particularly the sharing of knowledge. FLOSS are free for usage, modification and redistribution.

If you have already read this book, then you are familiar with FLOSS as well since you have been using **Python** all along!

If you want to know more about FLOSS, you can explore the following list. I have listed some big FLOSS as well as those FLOSS which are cross-platform (i.e. work on Linux, Windows, etc.) so that you can try using these software without the need to switch to Linux immediately *although you eventually will ;-)*

- **Linux.** This is a FLOSS operating system that the whole world is slowly embracing! It was started by Linus Torvalds as a student. Now, it is giving competition to Microsoft Windows. The latest 2.6 kernel is a major breakthrough w.r.t. speed, stability and scalability. [ Linux Kernel [http://www.kernel.org] ]

- **Knoppix.** This is a distribution of Linux which runs off just the CD! There is no installation required - you can just reboot your computer, pop the CD in the drive and start using a full-featured Linux distribution! You can use all the various FLOSS that comes with a standard Linux distribution such as running Python programs, compiling C programs, watching movies, etc. Then, reboot your computer again, remove the CD and use your existing OS, as if nothing happened at all. [ Knoppix [http://www.knopper.net] ]

- **Fedora.** This is a community-driven distribution, sponsored by Red Hat and is one of the most popular Linux distributions. It contains the Linux kernel, the KDE, GNOME and XFCE desktops, and the plethora of FLOSS available and all this in an easy-to-use and easy-to-install manner.

  If you care a complete beginner to Linux, then I would recommend that you try **Mandrake Linux** . The newly released Mandrake 10.1 is just awesome. [ Fedora Linux [http://fedora.redhat.com], Mandrake Linux [http://www.mandrakelinux.com] ]

- **OpenOffice.org.** This is an excellent office suite based on Sun Microsystems' StarOffice software. OpenOffice has writer, presentation, spreadsheet and drawing components among other things. It can even open and edit MS Word and MS PowerPoint files with ease. It runs on almost all platforms. The upcoming OpenOffice 2.0 has some radical improvements. [ OpenOffice [http://www.openoffice.org] ]

- **Mozilla Firefox.** This is **the** next generation web browser which is predicted to beat Internet Explorer (in terms of market share only ;-) in a few years. It is blazingly fast and has gained critical acclaim for its sensible and impressive features. The extensions concept allows any kind of functionality to be added to it.

  It's companion product Thunderbird is an excellent email client that makes reading email a snap. [ Mozilla Firefox [http://www.mozilla.org/products/firefox], Mozilla Thunderbird [http://www.mozilla.org/products/thunderbird] ]

- **Mono.** This is an open source implementation of the Microsoft .NET platform. It allows .NET applications to be created and run on Linux, Windows, FreeBSD, Mac OS and many other platforms as well. Mono implements the ECMA standards of the CLI and C# which Microsoft, Intel and HP have submitted for standardization and they have now become open standards. This is a step in the direction of ISO standardization for the same.

Currently, there is a complete C# **mcs** (which itself is written in C#!), a feature-complete ASP.NET implementation, many ADO.NET providers for databases and many many more features that are being improved and added everyday. [ Mono [http://www.mono-project.com], ECMA [http://www.ecma-international.org], Microsoft .NET [http://www.microsoft.com/net] ]

- **Apache web server.** This is the popular open source web server. In fact, it is **the** most popular web server on the planet! It runs nearly 60% of the websites out there. Yes, that's right - Apache handles more websites than all the competition (including Microsoft IIS) combined. [ Apache [http://www.apache.org] ]
- **MySQL.** This is an extremely popular open source database server. It is most famous for it's blazing speed. More features are being added to it's latest versions. [ MySQL [http://www.mysql.com] ]
- **MPlayer.** This is a video player that can play anything from DivX to MP3 to Ogg to VCDs and DVDs to ... who says open source ain't fun? ;-) [ MPlayer [http://www.mplayerhq.hu] ]
- **Movix.** This is a Linux distribution which is based on Knoppix and runs off the CD but is designed to play movies! You can create Movix CDs which are just bootable CDs and when you reboot the computer and pop in the CD, the movie starts playing by itself! You don't even need a hard disk to watch a movie using Movix. [ Movix [http://movix.sourceforge.net] ]

This list is just intended to give you a brief idea - there are many more excellent FLOSS out there, such as the Perl language, PHP language, Drupal content management system for websites, PostgreSQL database server, TORCS racing game, KDevelop IDE, Anjuta IDE, Xine - the movie player, VIM editor, Quanta+ editor, XMMS audio player, GIMP image editing program, ... this list could go on forever.

Visit the following websites for more information on FLOSS:

- SourceForge [http://www.sourceforge.net]
- FreshMeat [http://www.freshmeat.net]
- KDE [http://www.kde.org]
- GNOME [http://www.gnome.org]

To get the latest buzz in the FLOSS world, check out the following websites:

- OSNews [http://www.osnews.com]
- LinuxToday [http://www.linuxtoday.com]
- NewsForge [http://www.newsforge.com]
- SwaroopCH's blog [http://www.swaroopch.info/blog]

So, go ahead and explore the vast, free and open world of FLOSS!

# Appendix B. About

## Colophon

Almost all of the software that I have used in the creation of this book are *free and open source software*. In the first draft of this book, I had used Red Hat 9.0 Linux as the foundation of my setup and now for this sixth draft, I am using Fedora Core 3 Linux as the basis of my setup.

Initially, I was using KWord to write the book (as explained in the History Lesson in the preface). Later, I switched to DocBook XML using Kate but I found it too tedious. So, I switched to OpenOffice which was just excellent with the level of control it provided for formatting as well as the PDF generation, but it produced very sloppy HTML from the document. Finally, I discovered XEmacs and I rewrote the book from scratch in DocBook XML (again) after I decided that this format was the long term solution. In this new sixth draft, I decided to use Quanta+ to do all the editing.

The standard XSL stylesheets that came with Fedora Core 3 Linux are being used. The standard default fonts are used as well. The standard fonts are used as well. However, I have written a CSS document to give color and style to the HTML pages. I have also written a crude lexical analyzer, in Python of course, which automatically provides syntax highlighting to all the program listings.

## About the Author

Swaroop C H loves his job which is being a software developer at Yahoo! in the Bangalore office in India. His interests on the technological side include FLOSS such as Linux, DotGNU, Qt and MySQL, great languages like Python and C#, writing stuff like this book and any software he can create in his spare time, as well as writing his blog. His other interests include coffee, reading Robert Ludlum novels, trekking and politics.

If you are still to interested to know more about this guy, check out his blog at www.swaroopch.info [http://www.swaroopch.info] .

# Appendix C. Revision History

## Timestamp

This document was generated on January 13, 2005 at 00:05

Revision History

Revision 1.20 13/01/2005

Complete rewrite using Quanta+ on FC3 with lot of corrections and updates. Many new examples. Re-wrote my DocBook setup from scratch.

Revision 1.15 28/03/2004

Minor revisions

Revision 1.12 16/03/2004

Additions and corrections.

Revision 1.10 09/03/2004

More typo corrections, thanks to many enthusiastic and helpful readers.

Revision 1.00 08/03/2004

After tremendous feedback and suggestions from readers, I have made significant revisions to the content along with typo corrections.

Revision 0.99 22/02/2004

Added a new chapter on modules. Added details about variable number of arguments in functions.

Revision 0.98 16/02/2004

Wrote a Python script and CSS stylesheet to improve XHTML output, including a crude-yet-functional lexical analyzer for automatic VIM-like syntax highlighting of the program listings.

Revision 0.97 13/02/2004

Another completely rewritten draft, in DocBook XML (again). Book has improved a lot - it is more coherent and readable.

Revision 0.93 25/01/2004

Added IDLE talk and more Windows-specific stuff

Revision 0.92 05/01/2004

Changes to few examples.

Revision 0.91 30/12/2003

Corrected typos. Improvised many topics.

Revision 0.90 18/12/2003

Added 2 more chapters. OpenOffice format with revisions.

Revision 0.60 21/11/2003

Fully rewritten and expanded.

Revision 0.20 20/11/2003

Corrected some typos and errors.

Revision 0.15 20/11/2003

Converted to DocBook XML.

Revision 0.10 14/11/2003

Initial draft using KWord.